Technical Report 1080

A Concurrent Smalltalk Compiler for the Message-Driven Processor

Waldemar Horwat

MIT Artificial Intelligence Laboratory

A Concurrent Smalltalk Compiler for the Message-Driven Processor

Waldemar Horwat

Massachusetts Institute of Technology

May 1988

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. The research described in this paper was supported in part by Defense Advanced Research Projects Agency of the Department of Defense under contracts N00014-87-K-0825 and N00014-85-K-0124 and in part by a National Science Foundation Presidential Young Investigator Award with matching funds from General Electric Corporation and IBM Corporation.

A Concurrent Smalltalk Compiler for the Message-Driven Processor

by
Waldemar Horwat

Submitted to the Department of Electrical Engineering and Computer Science on May 6, 1988 in partial fulfillment of the requirements for the degree of Bachelor of Science in Electrical Science and Engineering

Abstract

This thesis describes Optimist, an optimizing compiler for the Concurrent Smalltalk language developed by the Concurrent VLSI Architecture Group. Optimist compiles Concurrent Smalltalk to the assembly language of the Message-Driven Processor (MDP). The compiler includes numerous optimization techniques such as dead code elimination, dataflow analysis, constant folding, move elimination, concurrency analysis, duplicate code merging, tail forwarding, use of register variables, as well as various MDP-specific optimizations in the code generator.

The MDP presents some unique challenges and opportunities for compilation. Due to the MDP's small memory size, it is critical that the size of the generated code be as small as possible. The MDP is an inherently concurrent processor with efficient mechanisms for sending and receiving messages; the compiler takes advantage of these mechanisms. The MDP's tagged architecture allows very efficient support of object-oriented languages such as Concurrent Smalltalk.

The initial goals for the MDP were to have the MDP execute about twenty instructions per method and contain 4096 words of memory. This compiler shows that these goals are too optimistic—most methods are longer, both in terms of code size and running time. Thus, the memory size of the MDP should be increased.

Thesis Supervisor: William J. Dally, Ph.D.
Title: Assistant Professor of Computer Science and Engineering

Keywords:

Compilation	Fine Grain	Object-Oriented Programming
Concurrent Smalltalk	Message-Driven Processor	Parallel Processing
CST	Message Passing	Programming Languages

Acknowledgements

I would like to thank

Professor Bill Dally for finding me at the beginning my sophomore year, getting me interested in the MDP project, guiding me through two semesters of UROPs, and providing unprecedented freedom, support, opportunities for growth, and exciting and important research opportunities to an undergraduate;

Brian Totty for writing an innovative operating system for the MDP and putting up with me when I requested numerous alterations in the operating system only to change my mind later and request different changes;

Scott Wills for sharing his hacking expertise and time;

Stuart Fiske for helping me get started with the Lisp Machines;

Ricardo Jenez for teaching me about compilers and helping me with problems that arose while I was designing the compiler;

Andrew Chien and Professor Dally for helping me with the intricacies of Concurrent Smalltalk and letting me use their Front End as part of my thesis;

Professor Bill Dally and Andrew Chien for creating and developing the Concurrent Smalltalk language without which this thesis would not have been possible; and

Professor Bill Dally, Andrew Chien, Soha Hassoun, Paul Song, Brian Totty, and Scott Wills for striving to make the MDP project successful.

Also, I would like to thank

Professor Jeffrey Lang, my advisor, for guiding me, helping me when I was lost, and being a friend; and

Bernard and Maria Horwat, my parents, for igniting in me the ambition that made me come to MIT while at the same not demanding too much of me; for sharing my joys and sorrows; and for letting me find my own way in the world.

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

The Optimist is an optimizing compiler for the Concurrent Smalltalk language. Concurrent Smalltalk is a concurrent version of Smalltalk developed by the Concurrent VLSI Architecture Group. The compiler compiles Concurrent Smalltalk to the assembly language of the Message-Driven Processor (MDP).

The Optimist includes some standard optimizations such as register variable assignment, dataflow analysis, copy propagation, and dead code elimination [2] [13] that are used in compilers for conventional processors. However, due to its fine-grained parallel nature, compiling for the MDP is unlike compiling for most conventional processors in a few important aspects which will become apparent in the later chapters. For instance, loops are not important on the MDP, while minimizing code size, tail forwarding methods, and efficiently and seamlessly handling parallelism are extremely important.

Several new optimizations or variants of optimizations were developed for the Optimist. For example, the Optimist includes Fork and Join Mergers that try to merge similar (not necessarily identical) statements on both sides of conditionals; often the Mergers successfully merge quite different statements, producing unusual (but nevertheless helpful) results. The Optimist's Move Eliminator is more powerful than standard copy propagation schemes such as the one given in [2]. While compiling, the Optimist works with a flow of control graph of statements and totally forgets the original order of statements in the source code, so it includes a Linearizer that tries to find the best linear order for the statements in the object code. Also, the Optimist includes numerous code generator optimizations to accomodate various idiosyncrasies of the MDP. Finally, several unexpected problems arise in the area of synchronizing processes through the use of futures; these problems and their solutions are presented in Chapter 6.

The MDP Project

The MDP is a processing node for the J-Machine, a multiple instruction/multiple data concurrent computer [6]. The J-Machine will be composed of up to 65536 MDPs. The nodes communicate with each other by sending messages over a high-speed network. The MDP nodes are optimized to minimize message sending and reception overheads; receiving and dispatching on a message or sending one should take only a few clock cycles, permitting efficient execution of finely grained concurrent programs. The project's goals are to fabricate the MDPs in VLSI, build a computer based on the MDPs, write the necessary operating system and language software, and analyze the performance of the resulting machine.

Concurrent Smalltalk

Concurrent Smalltalk is a concurrent version of the object-oriented programming language Smalltalk. It introduces concurrency to standard Smalltalk by evaluating arguments to method calls in parallel as well as allowing the computation of the value of a variable to proceed in parallel with the other computations of the method until the variable's value is actually needed. The `cset` construct (as opposed to `set`) is used to assign a value to a variable without actually requesting that the value be computed before going on to the next statement. For example, in the code sequence

```
(cset a (long-computation 1))
(cset b (time-sink 1))
(return (+ a b))
```

the computation of a proceeds in parallel with the rest of the method until the value of a is actually needed in the third statement; thus, long-computation and time-sink execute in parallel.

Please refer to the Chapter 2 for more details on the Concurrent Smalltalk language.

Why Concurrent Smalltalk?

Concurrent Smalltalk is an ideal language for programming the J-Machine because it yields small methods and locality of references. The methods dealing with a particular class travel to the data object as opposed to the data traveling to the code. Concurrent Smalltalk provides excellent facilities for creating data abstractions and "algorithm-independent programming"—once an algorithm to solve a problem has been developed for the J-Machine, application programs can use that algorithm as a library routine; furthermore, when an improvement in the algorithm is made, the change can be installed without rewriting application code.

Another reason for using Concurrent Smalltalk is that it is low-level enough to be useful in implementing the J-Machine operating system, while being at a high enough level that the programmer does not have to worry about the infamous problems of parallel process synchronization and deadlocks. In fact, once the data structures are defined properly, programming in Concurrent Smalltalk feels much like programming in a standard sequential language.

The Message-Driven Processor

The MDP is a processing node for the J-Machine. Each MDP chip contains a microprocessor, memory, and a network interface for communicating with neighboring MDPs.

The microprocessor has a register-based architecture. It operates on 32-bit data words with 4-bit tags that identify the type of the data. Tags are essential in efficiently supporting an object-oriented language such as Concurrent Smalltalk. Data types of variables in Concurrent Smalltalk are in general not known except when the program is actually running, so the compiler does not know whether a primitive operation such as + will receive integers or complex data structures like matrices as arguments. By implementing tags and type checking in hardware, the compiled code can just invoke the `add` instruction; if the arguments are not tagged as integers, the MDP will fault and the operating system will make a slower method call to add the matrices. Another reason for having tags is that they allow implementation of garbage collection algorithms that otherwise could not distinguish an integer from a pointer.

The MDP is message-based. In its normal mode of operation, the MDP listens on the network for messages. When it receives a message addressed to it, it stores the message in the input message queue and dispatches on the address given in the first word of the message. Messages are used as method calls; when a running process wants to call a method, it sends in the form of a SEND message the method selector together with the arguments onto the network, preferably to the node that contains the data object that is the receiver of the call. If the process is expecting a result, it includes a return address at the end of the message and stores a word tagged `CFUT` (context future) in its local variable that is to receive the result. That word marks that variable's value as unavailable until the method returns a value using a REPLY message; the returned value is written over the `CFUT`. Any access to an unavailable

variable will cause the process to wait until the value is available, thus providing automatic synchronization of data dependencies. While a process is waiting, other processes may run on the same node.

A detailed description of the MDP architecture is in [9]; a summary is presented in Appendix A. The formats of the various messages and the operating system interface is in Appendix B, while [12] contains a high-level and slightly obsolete description of the operating system. MDPSim [10] is an instruction level simulator, assembler, and debugger used to run MDP assembly language programs and test the operating system. It is also the current target environment of the compiler.

The Optimist Compiler

Goals

The main goal of the Optimist compiler is to produce Concurrent Smalltalk code that is as small as possible without sacrificing speed. In almost all cases optimizations that reduce space also reduce speed, but there are a few cases in which they conflict; in those cases the decisions were made in favor of optimizing space. The compiler does not make any optimizations if they would affect the semantics of the Concurrent Smalltalk program. There were a few cases (for instance, the inline coding of primitives) in which highly desirable optimizations could not be done due to little-used details of the specification of Concurrent Smalltalk; in those cases the semantics were modified to permit efficient compilation.

Compilation speed was not a major goal of the compiler project. Simplicity and flexibility were considered more important. Thus, there are numerous portions of the compiler that could be accelerated at an expense of simplicity and flexibility. Still, the compiler does achieve reasonable compilation speed, taking between one and fifteen seconds to compile most methods on a Macintosh[1] II.

Implementation

The Optimist compiler is written in Common Lisp. It adheres to standard Common Lisp as specified in [11] with the exception of using the LOOP iteration macro [3]. The LOOP macro is itself written in standard Common Lisp, so Optimist should run on any machine with a faithful implementation of Common Lisp.

Optimist was developed on a Macintosh using Allegro Common Lisp written by Coral Software Corp. and Franz Inc. It runs on a 2-megabyte Macintosh II. It was successfully tested on Sun Common Lisp developed by Sun Microsystems, Inc. and Lucid, Inc. and on a Symbolics 3600 workstation running Common Lisp.

Contribution to MDP Project

The Optimist compiler is part the language software part of the MDP project. Currently it allows execution and performance measurements of Concurrent Smalltalk programs on the J-Machine simulator [10]. In the future it will serve as the compiler for Concurrent Smalltalk programs for the actual J-Machine.

As will become apparent later, though, the compiler's effects on the MDP projects are more profound, as implementing the compiler did help solidify the system software, the MDP Architecture, and the definition of Concurrent Smalltalk. Moreover, an analysis of the code output by the compiler indicates that the original estimates of the amount of time it takes to

[1] Macintosh is a trademark of Apple Computer, Inc.

receive and process a message were too low, although it is not yet clear by how much. Furthermore, despite the optimizations, the size of the code output by the compiler was larger than was anticipated, thereby forcing a reconsideration of the amount of memory accessible to the MDP.

Outline

The next chapter, **Concurrent Smalltalk**, contains a quick introduction to the Concurrent Smalltalk language as well as the differences between the language as given in other sources and the Optimist implementation of it.

The **Compiler Overview** chapter follows. This chapter presents a basic overview of the compiler and introduces the major sections of the compiler. It is followed by more detailed chapters on each of the basic sections, roughly in the order in which data flows within the compiler. While reading the basic sections it might be helpful to refer to the **Examples** chapter and **Appendix C**. The **Examples** chapter examines the step-by-step compilation of a sample Concurrent Smalltalk method. It shows in a concrete example how the compiler's subsystems fit together and how they contribute to the final output. **Appendix C** contains the descriptions of the utilities used by the other compiler sections; understanding the utilities available to the other sections may be helpful in understanding what the other sections do.

There are many, many details that have to be considered in writing a compiler, and including them all in this thesis would make it exceedingly long (and boring). Thus, many aspects of the compiler's operations have been simplified or omitted even in the detailed descriptions. For the definitive information on how a particular subsystem works please refer to **Appendix D**, which contains the complete listing of the compiler. **Appendices A** and **B** are specifications of the compiler's assumptions about the target machine; the information there may be useful in understanding the compiler's output.

Appendix E is an Optimist user's manual and contains instructions for actually running the compiler.

Finally, the results and experiences with the compiler are given in the **Conclusion** chapter, which also contains recommendations for future work and improvements.

Chapter 2. Concurrent Smalltalk

The language Concurrent Smalltalk was developed by William Dally [5]. It is a concurrent version of the Smalltalk-80 language [8]. As extensions of Smalltalk-80, it includes the abilities to send messages without waiting for replies, concurrently access objects, and create objects that are distributed over the nodes of the machine. A recent description of the Concurrent Smalltalk language together with some examples is in [7].

Reality

The version of Concurrent Smalltalk supported by this compiler does not include distributed objects because they are not supported by the operating system [12]. Nevertheless, once distributed objects are added to the operating system, the changes to be made to the compiler will be minimal. Other features not yet supported due to lack of operating system support and a limited amount of time include block scoping and global variables.

Unlike the *infix* description of Concurrent Smalltalk in [5], the source code for the compiler uses the *prefix* format that looks like Lisp code. There is no semantic difference between the two formats, and they can be converted one-to-one into each other.

Syntax

The syntax of Concurrent Smalltalk accepted by the compiler is given in Table 2-1.

A program is a sequence of definitions. The definitions that are currently supported are `constant`, `class`, and `method` definitions. In addition, a file inclusion facility is provided by the `Load` statement, which includes the specified file at the point of the `Load` statement.

A `constant` definition defines a constant named <constant-name>. The constant can be either an integer or a named symbol. Once a constant is defined, it may not be redefined or changed. Constants encountered in methods are replaced by their values at compile time. Predefined constants are listed in Table 2-2.

A `class` definition defines a new Concurrent Smalltalk class. A class is a template for specifying objects and methods. Each object belonging to the class contains the instance variables defined in the class definition as well as the instance variables inherited from its superclasses, if any. If an instance variable is specified that has the same name as an instance variable of one of the superclasses, the new instance variable shadows the old one in the definitions of methods for the new class.

A few methods are automatically defined when a class is defined. Specifically, for each instance variable a method is defined with the same name as the instance variable that, when called on an object of the given class, returns the value of that instance variable. These methods are called *accessor* methods.

A `method` definition defines a method named <method-name> for class <class-name> and any classes derived from that class (unless that method is overridden by another method defined with the same name for a subclass). Each method is allowed zero or more formal arguments as well as zero or more local variables that exist for the duration of the method's execution. The names of these are specified in the method definition. The last item in the method definition is the definition of the actual method code, given as a series of expressions.

When a method is called, the values of the formals are computed and assigned to the formals. After all formals are computed, execution of the method's expressions proceeds as if the expressions were enclosed in an implicit `block`—initially the first expression is evaluated, then the second one, and so forth. The value of the implicit `block`, which is the value of the last expression, is returned to the caller unless an `exit` statement is encountered first.

Table 2-1. Concurrent Smalltalk Syntax

```
<program> ::=        <definition>*
<definition> ::=     (Constant <constant-name> <value>) |
                     (Class <class-name> (<superclass-name>*) <instance-var-name>*) |
                     (Method <class-name> <method-name> (<formal-name>*)
                           (<local-name>*) <expression>*) |
                     (Load "file-name")
<value> ::=          <integer> | <symbol-name>
<expression> ::=     <integer> |
                     (quote <symbol-name>) |
                     '<symbol-name> |
                     self |
                     <formal-name> |
                     <local-name> |
                     <instance-var-name> |
                     <constant-name> |
                     <method-name> |
                     (<method-expression> <receiver-expression> <expression>*) |
                     (set <target-name> <expression>) |
                     (cset <target-name> <expression>) |
                     (touch <expression>) |
                     (new <class-name>) |
                     (if <expression> <expression> [<expression>]) |
                     (begin <expression>*) |
                     (reply <expression>) |
                     (return <expression>) |
                     (exit)
<target-name> ::=    <local-name> | <instance-var-name>
```

Table 2-2. Predefined Constants

Constant	Value
T	TRUE
TRUE	TRUE
FALSE	FALSE
NIL	NIL

Expressions

As shown in Table 2-1, an expression is either a constant, a reference to a variable, a call, or one of the control constructs. Each expression returns a value that may be used or ignored.

The allowed constant expressions are integers, quoted symbols, names of previously defined constants, and method names. These all evaluate to their own values. Variable expressions may refer to formal, local, or instance variables, as well as `self`, which is the object on which the method was called. These expressions evaluate to the variables' current values.

A method call is specified by giving the method name followed by the receiver (the object to which the method is applied) as well as the method's arguments, if any. Called methods execute concurrently with the caller method unless some form of synchronization such as `set` or `touch` is used. The order of evaluation of arguments is not specified; in fact, some of them may be evaluated in parallel, and some may not be evaluated at all if they are not necessary. The method name does not have to be a constant; in fact, it can be any expression.

`Set` and `cset` assign values of expressions to variables which must be either instance variables or local variables (assignments to formals are not allowed). The value of the expression is the value assigned to the variable. The difference between `set` and `cset` is that `set` waits until the value is calculated before proceeding, while `cset` proceeds immediately, allowing the calculation of the value to execute concurrently with the calling method until the value is actually needed; if the value is not ready at that point, the calling method will wait until the value is available. This synchronization is transparent to the programmer. Thus, `csets` should be used wherever possible to improve performance (and decrease code size).

`Touch` is like `set` in that it evaluates the expression and waits until the value is available before proceeding. It returns the value of the expression.

`New` creates and returns a new object of the specified class. The object is not initialized.

`If` evaluates the first expression, which must return either `TRUE` or `FALSE`. If it returns `TRUE`, the second expression is evaluated and its value returned; otherwise, the third expression, if any, is evaluated and its value returned. If there is no third expression, the value is `NIL`. `If` does *not* wait until its value is available before returning.

`Begin` evaluates the expressions one by one and returns the value of the last one. `Begin` does *not* wait until its value is available before returning.

`Reply` evaluates its expression and replies the value of the expression to the caller of the current method. Execution then proceeds with the next statement of the current method, if any. `Exit` terminates the processing of the current method without sending a reply, which may cause the caller method to hang if it expected a reply. `Return` is like `reply` except that after replying the value of expression, it performs an `exit`. Although `return` is a safe statement, `reply` and `exit` should be used with caution, as `exit` may cause the caller to hang, while `reply` may cause the caller to crash if two replies are inadvertently sent. When using `reply` it is important to note that there is an implicit `reply` of the last expression in the method code that is always executed unless an `exit` is called first; thus, every explicit `reply` must be followed by an explicit `exit`.

Primitives

Primitive classes are provided for reasons of efficiency and convenience. Certain primitive operations on primitive classes are compiled into single assembly language instructions instead of method calls, improving their speed greatly. The four primitive classes are listed in Table 2-3.

Other primitive classes may be defined by methods written in assembly language and linked with the programs generated by the compiler. `Arrays` are defined in this way.

Certain method names are reserved as primitives that compile to assembly language instructions. These are listed in Table 2-4.

Table 2-3. Primitive Classes

Class	Values
Integer	Arbitrary-sized integers
Symbol	Symbols, including all symbolic constants and NIL, but not TRUE and FALSE
Boolean	The booleans TRUE and FALSE
Float	Floating-point numbers (not implemented in operating system)

Table 2-4. Primitive Methods

Name	Class	Number of arguments (including receiver)	Action
neg	Integer	1	Return -arg1.
+	Integer	0 or more	Return the sum of the integer arguments.
-	Integer	2	Return the difference of the two arguments.
*	Integer	0 or more	Return the product of the arguments.
//	Integer	2	Return arg1/arg2, rounding towards -∞. An error occurs if arg2=0.
mod	Integer	2	Return arg1-(arg1//arg2)*arg2. An error occurs if arg2=0.
ash	Integer	2	Return $arg1*2^{arg2}$, rounding towards -∞ if arg2 is negative.
min	Integer	1 or more	Return the smallest argument.
	Boolean	1 or more	Return the AND of the arguments.
max	Integer	1 or more	Return the largest argument.
	Boolean	1 or more	Return the OR of the arguments.
not	Boolean	1	Return the logical negation of the argument.
and	Boolean	0 or more	Return the logical AND of the arguments.
or	Boolean	0 or more	Return the logical inclusive OR of the arguments.
xor	Boolean	0 or more	Return the logical exclusive OR of the arguments.
lognot	Integer	1	Return the logical negation of the argument.
	Boolean	1	Return the bitwise complement of the argument.
logand	Integer	0 or more	Return the bitwise AND of the arguments.
	Boolean	1 or more	Return the logical AND of the arguments.
logor	Integer	0 or more	Return the bitwise inclusive OR of the arguments.
	Boolean	1 or more	Return the logical inclusive OR of the arguments.
logxor	Integer	0 or more	Return the bitwise exclusive OR of the arguments.
	Boolean	1 or more	Return the logical exclusive OR of the arguments.
<	Integer	2	Return TRUE if arg1<arg2 and FALSE otherwise.
	Boolean	2	Return (NOT arg1) AND arg2.
<=	Integer	2	Return TRUE if arg1≤arg2 and FALSE otherwise.
	Boolean	2	Return (NOT arg1) OR arg2.
>	Integer	2	Return TRUE if arg1>arg2 and FALSE otherwise.
	Boolean	2	Return arg1 AND (NOT arg2).
>=	Integer	2	Return TRUE if arg1≥arg2 and FALSE otherwise.
	Boolean	2	Return arg1 OR (NOT arg2).
=	Integer	2	Return TRUE if arg1=arg2 and FALSE otherwise.
	Boolean	2	Return TRUE if arg1=arg2 and FALSE otherwise.
	Symbol	2	Return TRUE if arg1=arg2 and FALSE otherwise.
<>	Integer	2	Return TRUE if arg1≠arg2 and FALSE otherwise.
	Boolean	2	Return TRUE if arg1≠arg2 and FALSE otherwise.
	Symbol	2	Return TRUE if arg1≠arg2 and FALSE otherwise.
eq	Any combination	2	Return TRUE if arg1=arg2 and FALSE otherwise.
neq	Any combination	2	Return TRUE if arg1≠arg2 and FALSE otherwise.

Eq and neq are pointer comparisons and cannot be redefined.

The rationale for having // and mod round towards -∞ is that this definition allows the use of arithmetic shifts to divide and logical AND to find the remainder when the divisor is an integral power of two.

If a primitive method is called with an argument that is not one of the primitive classes it recognizes, the actual method for the class is found and executed (this is not yet implemented in the operating system). Thus, it is possible to define a class of type, say, complex, and define a method * for numbers of that type. That method will be called whenever * is used on a number of type complex (regardless of whether that number is the receiver or the argument).

It should be noted, though, that since the compiler generates assembly language instructions for the primitive methods instead of method calls, methods overriding primitives must satisfy certain identities which are listed in Table 2-5.

Table 2-5. Identities among Primitive Methods

- + is associative and commutative.
- 0 is an identity for +.
- (- a b) = (+ a (neg b)).
- * is commutative with scalar constants and associative.
- 1 is an identity for *.
- (* a 2^e) = (ash a e).
- (// a 2^e) = (ash a -e).
- (ash 0 a) = 0.
- (ash a 0) = a.
- min and max are associative and commutative.
- (not (not a)) = a.
- and, or, and xor are associative and commutative.
- (and a FALSE) = FALSE.
- (and a TRUE) = a.
- (or a FALSE) = a.
- (or a TRUE) = TRUE.
- (xor a FALSE) = a.
- (xor a TRUE) = (not a).
- (lognot (lognot a)) = a.
- logand, logor, and logxor are associative and commutative.
- (logand a 0) = 0.
- (logand a -1) = a.
- (logor a 0) = a.
- (logor a -1) = -1.
- (logxor a 0) = a.
- (logxor a -1) = (lognot a).
- (< a b) = (not (>= a b)).
- (> a b) = (not (<= a b)).
- (= a b) = (not (<> a b)).
- (< a b) = (> b a).
- (<= a b) = (>= b a).
- (= a b) = (= b a).
- (<> a b) = (<> b a).

These identities have been carefully selected to allow efficient implementation of primitive operations without sacrificing functionality. Some identities have been omitted on purpose. For example, * does not have to be commutative in general, nor does (* a 0) have to equal 0. Not requiring these identities allows * to be used to multiply quaternions and matrices.

Chapter 3. Compiler Overview

The compiler is organized into several sections which perform a series of transformations on the code. These sections are illustrated in Figure 3-1. The Front End and its library handlers were originally written by Prof. William Dally and Andrew Chien; I made modifications to them to adapt them to this compiler, fix a few minor problems, and improve the syntax of Concurrent Smalltalk. Everything else is entirely my own.

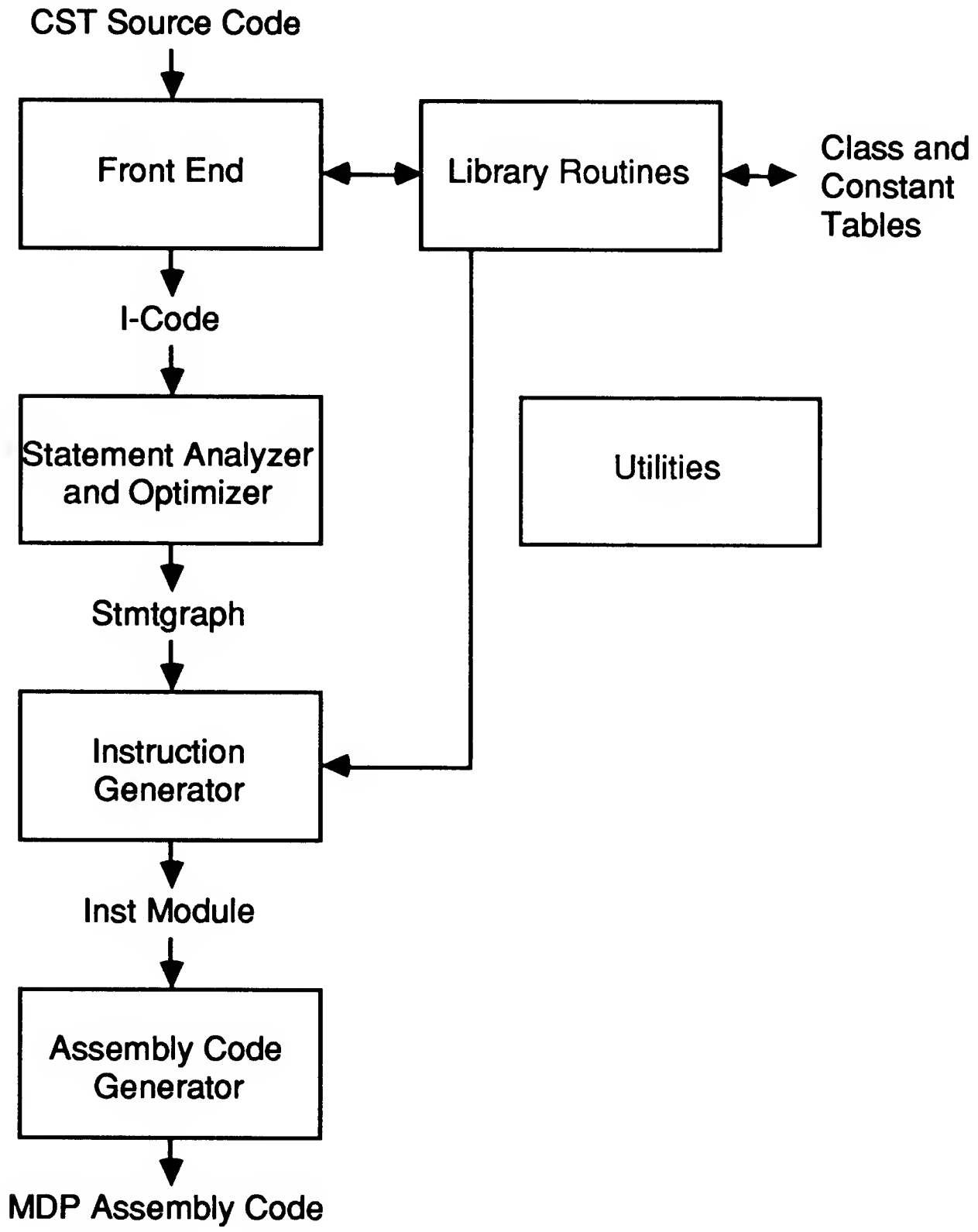


Figure 3-1. Compiler Block Diagram.

Utilities

There are a few Lisp data types and functions that are used throughout the compiler. These functions include an efficient implementation of general sets of nonnegative integers, directed graphs (digraphs), and generalized algorithms such as mapping, basic block-finding, and calculating dataflow information by relaxation on digraphs. These functions and data types have been collected in the Utilities file and are described in Appendix C.

Front End

The source code is converted by the Front End into a language called I-Code that is similar to the "quadruples" code that many compilers use. The I-Code is at a somewhat higher level than the quadruples code, though, in that it specifies units such as entire procedure calls in single instructions. The I-Code also allows for the possibility of having more than one source language compile into MDP assembly language code or having the same source language compile into several assembly languages. The syntax of the I-Code is given in Table 3-1.

The library handler is really part of the Front End. Its main function is keeping track of the classes and constants that have been defined. As such, it is used as a subroutine by other blocks that would like to know information about classes.

Table 3-1. I-Code Syntax

```
<method> ::=      <statement>*
<statement> ::=   (Csend <target> <selector-slot> <receiver-slot> <argument-slot>*) |
                  (Touch <source-slot>) |
                  (Move <target> <source-slot>) |
                  (New <target> <class-name>) |
                  (Reply | Reply-x <slot>) |
                  (Return | Return-x <slot>) |
                  (Label <label>) |
                  (Jump <label>) |
                  (Falsejump <slot> <label>)
<target> ::=      (temp <name>) |
                  (var <name>) |
                  (ivar <number>)
<slot> ::=        (temp <name>) |
                  (var <name>) |
                  (ivar <number>) |
                  (arg <number>) |
                  (const <constant>) |
                  (method <name>) |
                  self
<constant> ::=    <integer> | <symbol> | NIL | T | True | False
```

The correspondence between most I-Code statements and their Concurrent Smalltalk counterparts is straightforward. The only differences worth noting are that all function calls compile to `csends` and that a Concurrent Smalltalk `set` does an automatic `touch` on its target before proceeding. `Reply-x` and `return-x` will be used to implement block scoping and currently perform the same function as `reply` and `return`.

Statement Analyzer and Optimizer

The Statement Analyzer and Optimizer processes the I-Code generated by the Front End to produce a stmtgraph. The stmtgraph is a directed graph implemented with the digraph data type defined in Utilities in which each node represents an I-Code statement (stmt) and each edge represents a possible flow of control path from one statement to another. The digraph contains no unconditional branches, as these are represented simply by connecting the predecessor statement to the successor statement with an edge. The statements allowed in the digraph nodes are listed in Table 3-2. They are similar to the I-Code statements, but there are a few differences.

Table 3-2. Possible Stmts

Operation	Target	Method	Arguments
`enter`			
`csend`	dest-slot		(selector-slot receiver-slot arg1-slot ... argn-slot)
`rsend`			(selector-slot receiver-slot arg1-slot ... argn-slot)
`primitive`	dest-slot	primitive	(receiver-slot arg1-slot ... argn-slot)
`move`	dest-slot		(source-slot)
`touch`			(source-slot)
`new`	dest-slot	class-name	
`condition`		condition	(source-slot)
`reply`			(source-slot)
`exit`			

Any fields that are blank are set to `NIL`. The `primitive` statement is a method call with a selector that was recognized as one of the primitives listed in Table 2-4. The `enter` statement is placed at the beginning of the method and performs some initialization functions. The `condition` statement corresponds to one of the six of the MDP's conditional branches; condition specifies the type of branch, which may be one of the following:

`bnil`	Branch if source-slot is `eq` to `NIL`
`bnnil`	Branch if source-slot is not `eq` to `NIL`
`bf`	Branch if source-slot is `false`
`bt`	Branch if source-slot is `true`
`bz`	Branch if source-slot is = to `0`
`bnz`	Branch if source-slot is <> to `0`

The `rsend` statement is a tail-forwarded `csend`—the result of the `rsend` is sent to the caller of this method instead of this method. The target of a `csend` may be `NIL`, in which case the return value of the `csend` is ignored.

The Statement Analyzer and Optimizer performs all of the compiler's optimizations that are relevant at the I-Code level of abstraction. These optimizations include dead code elimination, move elimination, dataflow transformations, constant folding, tail forwarding, and merging of identical statements on both sides of forks and joins. A fork is a statement with more than one outgoing flow of control path; currently `conditions` are the only forks. A join is a statement with more than one incoming flow of control path.

The stmtgraph produced by the Statement Analyzer and Optimizer can be converted back into a modified version of plain I-Code. There is a function in the Statement Analyzer and Optimizer, output-stmtgraph, available to do this conversion; the function is useful for the purposes of debugging as well as using the Statement Analyzer and Optimizer to optimize I-Code that will be run on Andrew's simulator [7].

Instruction Generator

The Instruction Generator converts the stmtgraph into a module, which is another digraph. The nodes of a module represent individual MDP assembly language instructions as opposed to I-Code statements. As in the stmtgraph, the edges of a module correspond to flow of control paths through the instructions. There are no unconditional branches in the resulting module. However, unlike in the stmtgraph, the instructions in the module are ordered in a linear sequence that represents their order in the final assembly language output.

An important function performed by the Instruction Generator before it generates code is assigning variables to locations. Each local variable and temporary can be assigned to either a register or a slot in the context object. The Instruction Generator tries to assign as many variables as possible to registers and use as few context slots as possible, and it will reuse registers and context slots whenever possible.

The Instruction Generator performs statement-specific optimizations on I-Code statements. It also keeps track of the values of the MDP registers while it is compiling, allowing it to use values in registers whenever available. The Instruction Generator does not, however, perform any final peep-hole optimizations on the module.

Assembly Code Generator

The Assembly Code Generator inserts branches into the module created by the Instruction Generator and performs several peep-hole optimizations on that module. The important optimizations include shifting instructions wherever possible to align DC instructions to word boundaries and combining SEND and SENDE instructions to SEND2 and SEND2E. The Assembly Code Generator also checks each branch to make sure that the branch destination is reachable from the branch source within the limited MDP branching range; if not, the branch is replaced by a long branch. This process also involves several optimizations.

Finally, the Assembly Code Generator outputs the module as a series of assembly language statements. The resulting file can be read, assembled, and executed by MDPSim, and, hopefully, eventually by a working J-Machine.

Chapter 4. Front End

The organization of the Front End is fairly straightforward. Since the prefix Concurrent Smalltalk code is in the form of lists readable by the Lisp reader, there is no need for a parser—the Front End routines accept list structures read from the input file. There are three routines in the Front End that are called from the outside: compile-class, compile-method, and instance-vars. The Front End also maintains three global lists: *classes*, a list of all defined classes, their superclasses, and their instance variables; *constants*, a list of all defined constants and their values, and *globals*, a list of all defined globals and their values. Globals are not implemented by the code generator, though, because the operating system does not provide a facility for them.

```
;; (class name ({parent-classes}) {instance-variables})
(defun compile-class (form output-stream)
 (let ((class (expand-class (cdr form))))
  (setq *classes* (cons class *classes*))
  (if output-stream (make-accessor-methods class output-stream))
  class))
```

Compile-class compiles a class definition—it checks the class definition for validity; calculates the class's instance variables by concatenating the instance variables of the superclass, if any, with the new instance variables; adds the new class to *classes*; compiles the accessor methods for the class; and outputs the resulting code onto a stream.

```
;;  (method class method-name ({args}) ({temps}) {statements})
(defun compile-method (form &optional (output-stream t))
 (if (< (length form) 6)
  (cst-error "~&Method missing field ~S" form)
  (let ((class-name (second form))
     (method-name (third form))
     (args (fourth form))
     (vars (fifth form))
     (body (nthcdr 5 form)))
   (let ((icode (compile-block args vars (instance-vars class-name) body)))
    (if output-stream
     (compile-icode method-name class-name (length args) icode :output-stream output-stream))
    icode))))
```

Compile-method is the general Front End routine for compiling a method. It takes a Lisp list that is the definition of the method and a stream onto which the assembly code for the compiled method should be written. It then calls compile-block to generate the I-Code for the method and, if the output stream is non-`nil`, compile-icode to compile the I-Code to assembly language. Finally, compile-method returns the I-Code as a help in debugging.

Compile-block sets up a few dynamic variables (see Appendix D for details) and compiles the statements of the method using compile-expression.

```
;;; compiles an expression and puts the result in slot
;;; if slot is nil, doesn't put the result anywhere.
;;; if slot is '_unbound_ creates a temporary
(defun compile-expression (slot expr)
 ;(format *standard-output* "~&compile-expression ~S ~S" slot expr)
 (if (atom expr)
  (compile-atom slot expr)
  (let ((head (car expr)))
   (if (eq (symbol-type head) 'keyword)
    (ecase head
        ((set cset) (compile-set slot expr))
        ((return return-x) (compile-return head slot expr))
        (reply
         (if *anachronisms* (compile-return 'return slot expr)
           (compile-reply 'reply slot expr)))
        (reply-x
         (if *anachronisms* (compile-return 'return-x slot expr)
           (compile-reply 'reply-x slot expr)))
        (forward ;anachronism
         (if (eq (cadr expr) 'requester)
          (compile-reply 'reply-x slot (list 'reply-x (cddr expr)))
          (cst-error "~&Can't reply to ~S" (cadr-expr))))
        (exit (emit '(exit)) slot)
        (iftrue (compile-iftrue slot expr))
        (if (compile-if slot expr))
        (begin (compile-begin slot expr))
        (new (compile-new slot expr))
        (newco (compile-newco slot expr))
        (quote (check-binding slot `(const ,(cadr expr))))
        (msg (compile-message slot expr))
        (send ;anachronism
         (compile-expression slot (cdr expr)))
        (touch (compile-touch slot expr)))
    (compile-send slot expr)))))
```

Compile-expression takes two parameters: a slot into which the value of the expression is to be stored, and the source code for the expression to be compiled. If the slot is the symbol _unbound_, compile-expression creates a new slot and stores its value there. Compile-expression returns the slot into which the value of the expression was actually stored. Numerous routines are called by compile-expression, one for each type of Concurrent Smalltalk source statement. They are all quite straightforward; please refer to Appendix D for their listings. The only two routines that may need additional explanation are symbol-type and check-binding.

```
(defun symbol-type (expr)
 (cond ((numberp expr) `(const ,expr))
     ((not (symbolp expr)) nil)
     ((eq expr 'self) 'self)
     ((eq expr 'super) 'super)
     ((eq expr 'group) 'group)
     ((member expr *vars*) `(var ,(index expr *vars*)))
     ((member expr *args*) `(arg ,(index expr *args*)))
     ((member expr *ivars*) `(ivar ,(index expr *ivars*)))
     ((symbol-is-keyword? expr) 'keyword)
     ((assoc expr *globals*) `(global ,expr))
     ((assoc expr *constants*) `(const ,(cdr (assoc expr *constants*))))
     (t (list 'method expr))))
```

Symbol-type returns the slot corresponding to a token read from the source code or the symbol keyword if the token is one of the Concurrent Smalltalk keywords. It implements a limited form of lexical scoping by checking for the local definitions before the global definitions; thus, a local definition of a variable may shadow the global definition of a constant or even a

Concurrent Smalltalk keyword. If none of the other definitions fits, symbol-type assumes that the symbol read is the name of a method.

```
;;; if a is already bound move b to a and return a otherwise return b
(defun check-binding (a b)
 (if (eq a '_unbound_)
  b
  (if (equal a b)
   a
   (progn (if a (emit `(move ,a ,b)))
        a))))
```

Check-binding is used by most of the I-Code-generating routines to place the result in the correct slot. Many of the routines place their results in temporary slots and later use check-binding to try to match the temporary slot (b) with the slot in which the enclosing statement expects the value (a). If the enclosing statement does not want to receive the result of the statement, it will set a to `nil`. If it does want the result but does not care about where the result should be, it will set a to '_unbound_, and check-binding will return the location of the result that will subsequently be returned to the routine compiling the enclosing statement.

Chapter 5. Statement Analyzer and Optimizer

The Statement Analyzer and Optimizer is divided into several sections as shown in Figure 5-1.

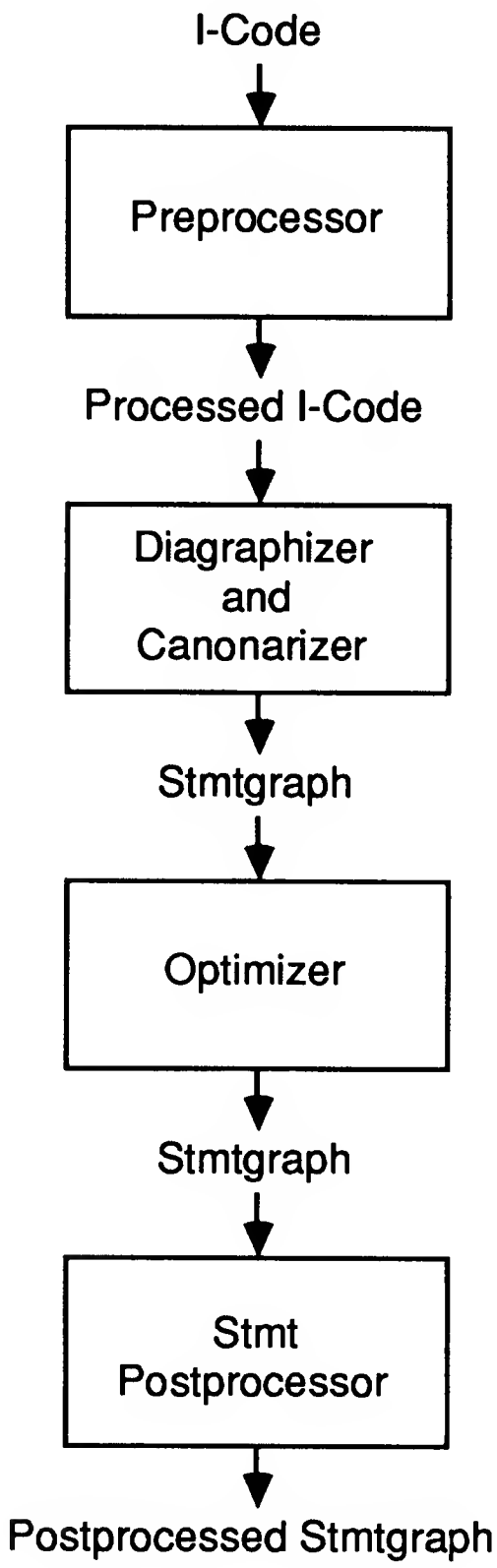


Figure 5-1. Statement Analyzer and Optimizer Block Diagram.

The preprocessor performs a few minor transformations to the I-Code. Its output is still I-Code. That I-Code is passed through the diagraphizer and canonarizer to produce a stmtgraph. The stmtgraph is then optimized by the optimizer, and some final transformations are done by the postprocessor.

Preprocessor

The I-Code preprocessor performs the following functions:

- It adds an `enter` statement to the beginning of the I-Code. The `enter` statement will later compile into initialization code for the method.

- It changes all existing `exit` statements into branches to one `exit` statement at the end of the method. That `exit` statement will later compile into termination code for the method, and having only one `exit` statement may save a few instructions.

- It changes all `return` statements into `moves` of the return value to a new variable and branches to one `reply` statement at the end of the method that then falls into the `exit` statement. Again, merging all of the `return` statements is likely to save some code.

Diagraphizer and Canonarizer

The diagraphizer converts the I-Code to a stmtgraph. It does this in two stages. First, it calls the routine diagraphize to translate the I-Code structure into a stmtgraph. Diagraphize scans the I-Code, replaces all branches in the I-Code with edges in the stmtgraph, and removes all labels from the code. Next, all of the slots in the stmtgraph are converted into the form used by the rest of the compiler. The actual syntax of the slots is listed at the beginning of the Stmt file in Appendix D. The slots representing local variables and temporaries generated by the Front End are merged into one category, *variables*, and assigned consecutive variable numbers starting with zero.

The preprocessor, diagraphizer, and canonarizer are all invoked, one after another, by the function input-icode, which also removes any dead code left over by disconnecting unreachable stmtgraph nodes using purge-unreachables-digraph (See Appendix C).

Statement Optimizer

```
;;;Perform iterative stmtgraph optimizations until a steady state is reached.
;;;Return the stmtgraph.
(defun iterative-optimize-stmtgraph (stmtgraph)
 (attribute-steady-state
  (stmtgraph-attributes stmtgraph)
  (progn
   (when *delete-dead-defs* (delete-dead-defs stmtgraph))
   (when *delete-moves* (delete-moves stmtgraph))
   (when *delete-touches* (delete-touches stmtgraph))
   (when *dflow-optimizations* (calc-dflow stmtgraph))
   (when *fold-constants* (fold-constants stmtgraph))
   (when *forward-sends* (forward-sends stmtgraph))
   (fold-conditionals stmtgraph) ;This must not be disabled, or code generator will fail!
   (when *merge-code*
    (merge-joins stmtgraph)
    (merge-forks stmtgraph))))
 stmtgraph)
```

The Statement Optimizer repeatedly tries a number of optimizations on the stmtgraph until none of them changes the stmtgraph. At that point it returns the stmtgraph. The optimizations attempted are listed in Figure 5-2 and in the listing above. Most optimizations can be disabled by setting the appropriate parameters to `NIL`. The optimizations are described in more detail below.

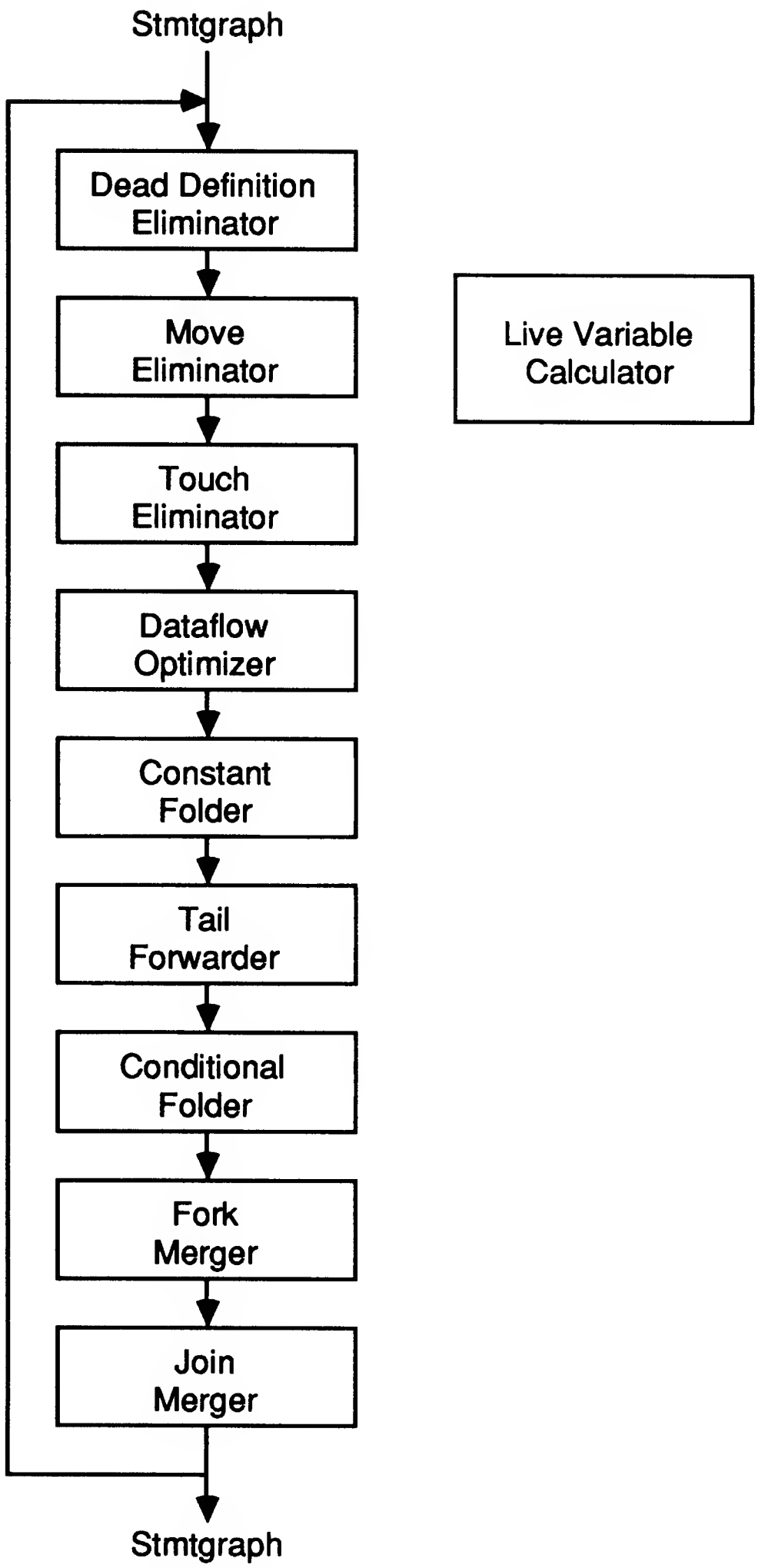


Figure 5-2. Statement Optimizer Block Diagram.
The Statement Optimizer calls the above optimization routines until a steady state is reached in which none of the optimization routines changes the stmtgraph.

Dead Definition Eliminator

This procedure removes all attempts to store values into dead variables; a variable is *dead* at a particular statement if that variable's value at that statement is not used by any statement that may subsequently execute. Variables that are not dead are called *live*. The Dead Definition Eliminator proceeds to eliminate stores to dead variables by first using micro-relax described in Utilities to calculate which variables are dead and which are live at every statement in the stmtgraph. It then scans the entire stmtgraph looking for stmts whose targets are dead local variables. Only `csend`, `primitive`, `move`, and `new` statements can be found, since only those stmts can have non-null targets (see Table 3-2). If the statement is a `primitive`, `move`, or `new`, it is removed, as removing it does not change the semantics of the program other than perhaps not causing an error that would otherwise occur. `Csends` cannot be safely removed; instead, if a `csend` has a target that is dead, the target is set to `NIL`, which will have the effect of asking the called object not to reply—still an improvement over `csending` to a dead variable.

Move Eliminator

The Move Eliminator attempts to remove as many `move` statements as possible from the stmtgraph. It works by scanning the entire stmtgraph looking for `move` statements. If it finds a `move` statement with an identical source and destination slot, that `move` statement is removed since it does not do anything. If the `move` statement moves a local variable to another local variable, the Move Eliminator tries to merge the two local variables by renaming one of them. Before doing the merge it checks whether the variables are simultaneously live at any point in the stmtgraph; if so, the merge cannot be safely performed, and the Move Eliminator abandons trying to optimize the `move`. Otherwise, the two local variables are merged, and the `move` statement removed.

The Move Eliminator complements the Dataflow Optimizer. Although they both try to optimize `move` statements, each is able to handle cases that the other cannot. The Move Eliminator's optimizations are restricted to `moves` with identical source and destinations and `moves` between two local variables. It optimizes these two cases quite well, though. On the other hand, the Dataflow Optimizer can eliminate `moves` from a constant, an argument, or an instance variable to a local variable, but with somewhat less flexibility. Figure 5-3 shows an example of `move` statements that cannot be eliminated by the Dataflow Optimizer yet which are easily handled by the Move Eliminator.

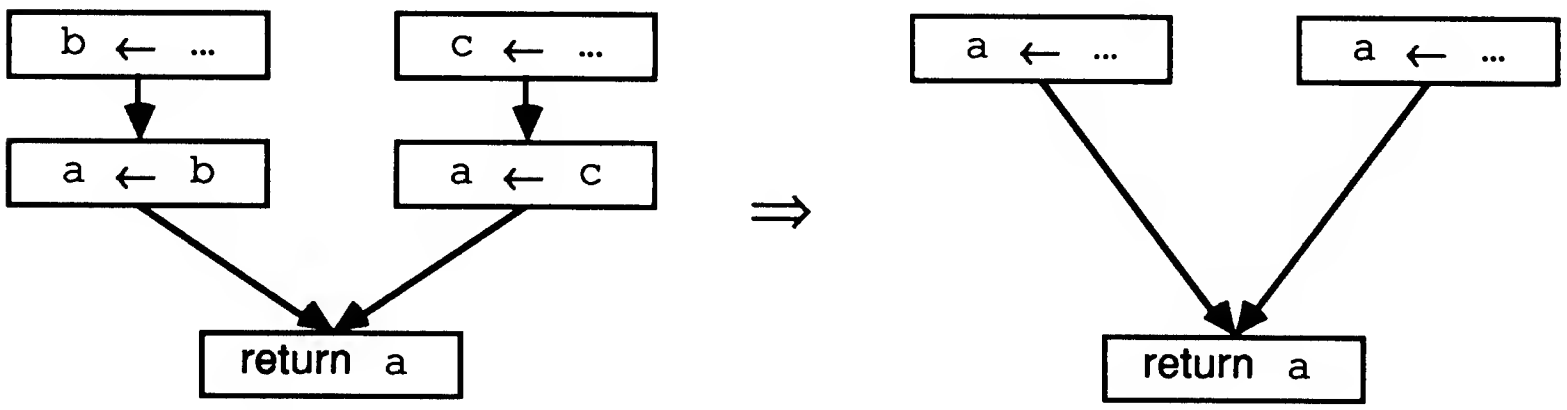


Figure 5-3. Move Eliminator Example.

The Move Eliminator is able to remove the two `move` statements (a←b) and (a←c) in the above example. The copy propagation algorithm used by the Dataflow Optimizer would not detect the opportunity to remove these two `move` statements because the value of `a` at the `return` statement is neither a copy of `b` nor a copy of `c`. This advantage of the Move Eliminator has great practical significance—the above example actually does occur in many methods.

Touch Eliminator

The Touch Eliminator eliminates all `touch` statements that it can prove are superfluous. A `touch` statement checks that its argument's value is available; if not, it waits until the value is available. Currently only local variables can contain values that are unavailable, so all `touch` statements referring to slots other than local variables are eliminated.

Eliminating `touch` statements that do refer to local variables is harder, but there are a few cases in which it is safe to remove the `touch` statements. For instance, if it can be shown (using relax-digraph to keep track of which variables are guaranteed to have available values even after some of the later optimizations are performed) that a local variable's value is always available when the `touch` statement is executed, the `touch` statement can be safely removed. Other examples of `touch` statements that can be removed are `touches` immediately preceding an `exit` statement (the `exit` statement touches all local variables to make sure that their values are available anyway, so the any touch statements immediately preceding it are superfluous), and `touches` immediately preceding `csend`, `rsend`, `touch`, and `reply` statements that refer to the variable that is touched, as these statements will always wait for the value of the variable anyway.

Dataflow Optimizer

The Dataflow Optimizer uses relax-digraph to perform a general analysis of the possible values that each local variable could have at all statements in the method. It is an extension of the copy propagation algorithms used in compilers such as the one described on page 637 of [2]. The algorithm makes one of the assertions listed in Table 5-1 about each local variable at every statement in the stmtgraph. If the Dataflow Optimizer cannot determine which assertion in Table 5-1 holds, if more than one of them holds (other than `NIL`), or if none of them holds, the `NIL` assertion is conservatively assumed.

Table 5-1. Dataflow Assertions about Local Variables

Assertion	Meaning
`NIL`	None of the other assertions holds.
`(move <slot>)`	A copy of <slot> (which may be a constant).
`(not <slot>)`	The primitive `not` applied to <slot>.
`(= <slot1> <slot2>)`	The primitive = applied to <slot1> and <slot2>.
`(<> <slot1> <slot2>)`	The primitive <> applied to <slot1> and <slot2>.
`(eq <slot1> <slot2>)`	The primitive `eq` applied to <slot1> and <slot2>.
`(neq <slot1> <slot2>)`	The primitive `neq` applied to <slot1> and <slot2>.

The Dataflow Optimizer then examines each statement in the stmtgraph and checks whether any local variable whose value is used by that statement has a non-`NIL` assertion. If so, then it tries to substitute the assertion into the statement. The `move` assertion can always be substituted—the `move` assertion's <slot> is substituted instead of the local variable. This copy propagation is the way constant `move` statements are eliminated—if a constant is moved into a local variable, then the Dataflow Optimizer replaces all references to that variable with the constant itself, and the Dead Definition Eliminator is then able to eliminate the `move` statement because its value is not used by any statement.

The other assertions are substituted only in special circumstances. The `not` assertion is substituted into another `not` primitive to yield a `move` statement: `(not (not x)) = x`. Similarly, `not` can be merged with `eq` to make a `neq` statement, `neq` to `eq`, = to <>, and <> to =.

However, the principal reason for keeping track of the `not`, `=`, `<>`, `eq`, and `neq` assertions is for their use in conditional branches. Statements such as

```
(if (= a 0) ...) or (if (not (eq a nil)) ...)
```

occur frequently in Concurrent Smalltalk code, and they should not generate calls to the `=`, `eq`, and `not` primitives because these operations can be done with conditional branches. Therefore, if the Dataflow Optimizer encounters a `bt` or `bf condition` (see Table 3-2), and if there is an assertion about the `condition`'s source slot, then it does the following:

- If the assertion is `not`, the `condition`'s source slot is replaced with the assertion's slot and the meaning of the branch is reversed.

- If the assertion is `=` or `<>` and one of the assertion's slots is `0`, the `condition`'s source slot is replaced with the assertion's other slot and the condition type changed to `bz` or `bnz`.

- If the assertion is `eq` or `neq` and one of the assertion's slots is `nil`, the `condition`'s source slot is replaced with the assertion's other slot and the condition type changed to `bnil` or `bnnil`.

In all of the above cases the Dataflow Optimizer does not remove the intermediate `not`, `=`, `<>`, `eq`, and `neq` statements that may no longer be needed. Instead, it relies on the Dead Definition Eliminator to eliminate them because their results are no longer used. If it turns out that the intermediate values generated by `not`, `=`, `<>`, `eq`, or `neq` are actually used somewhere else, the Dead Definition Eliminator will not eliminate these intermediate statements, and the Dataflow Optimizer might have increased the code size slightly, but this case does not occur often.

Constant Folder

Although the Dataflow Optimizer may substitute constants into primitive statements, it does not simplify the resulting statements. For example, as a result of a substitution, a primitive statement that adds `a` and `b` might be changed to a primitive statement that adds `1` and `3`. The Constant Folder's task is to simplify primitive constant expressions as far as possible. It knows the rules in Table 2-5 and applies them to primitive statements. For example, it converts `(primitive b + 0 a)` into `(move b a)`. It is capable of collecting constants together, so `(primitive b + -3 a 7 -4)` is also converted to `(move b a)`, which may be later eliminated by the Move Eliminator.

The Constant Folder's optimizations are not limited to `primitives`. It also examines `conditions` and checks whether they would always branch one way (i.e. if `bt` has an argument that is `true` or `false`, if `bnz` is invoked on `3`, `bnil` invoked on a symbol, etc.). If so, then the condition is removed, as is the stmtgraph's "dead" flow of control edge originating from the condition. Since removing an edge may produce dead code, the Constant Folder finally calls purge-unreachables-digraph on the stmtgraph to make sure that any new dead code is disconnected from the stmtgraph.

Tail Forwarder

The tail forwarder produces the MDP's equivalent of tail recursion. It is often the case that the value returned by a Concurrent Smalltalk method is the value returned by the last statement of that method, and that statement is often a method call. An example of this phenomenon is an iterative definition of the factorial function such as

```
(Method integer factorial (n) ()
  (if (= self 0) n (factorial (- self 1) (* n self))))
```

If self is not equal to zero, the factorial method calls factorial and immediately returns the result. There is, however, no fundamental reason why factorial should wait for the result of the recursive call to factorial only to return it to the caller; on the contrary, it would be better if it could just tell the recursive factorial call to return its result to the caller. This way the returning process would be significantly faster, and, more important, factorial can deallocate its context and cease execution as soon as it sends the recursive call to factorial. This way factorial runs in constant space (at least until the numbers get too large to fit in a word) as opposed to space proportional to n because the contexts of tail-forwarded factorials do not have to be stored.

The operation of the Tail Forwarder is simple. The Tail Forwarder scans the stmtgraph looking for `csend` statements that store their results in local variables. If it finds such a `csend` statement, it checks whether the statement following it is a `reply` statement of the same variable and that variable is dead afterwards. If so, the Tail Forwarder changes the `csend` statement into an `rsend` and connects it to the statement following the `reply` statement.

Conditional Folder

The Conditional Folder is a very simple optimization. It scans the stmtgraph for conditionals both of whose branches point to the same statement. Any such conditionals are removed. Although such conditionals do not normally appear in source Concurrent Smalltalk code, they can be created as a result of some other optimizations such as the implicit dead code elimination in the Diagraphizer, the Reply Forwarder, and Fork and Join Mergers.

Fork and Join Mergers

These two optimizations, if they can be applied, often produce significant savings in the output code size. They try to consolidate similar statements on both sides of forks (conditionals) and joins (places where two paths of control flow merge) in the stmtgraphs. Currently they only consider the first statements after the forks or before the joins, but they can be extended to consider other statements as well.

The Fork Merger considers every conditional in the stmtgraph. For each conditional it checks whether the statements following it are of the same type (both are `csends`, `rsends`, the same kind of `primitive`, `reply`, etc.). If the types match, if the statements' arguments (but not necessarily targets) are identical, if there are no flow-of-control edges other than from the conditional entering either of the statements, and if neither of the statements writes to the variable used by the conditional statement, then the two statements after the conditional are merged into one statement before the conditional. If the targets of the two statements were originally different, then the new statement before the conditional writes its result into a temporary variable, and two `move` statements from the temporary variable to the two variables where the result would have gone are placed after the conditional. These `move` statements are often later eliminated by the Move Eliminator.

The Join Merger operates in a manner similar to the Fork Merger except that it does not have to worry about interaction with the condition variable because there is none. For two statements to be considered by the Join Merger to be similar, they have to have identical targets and the same number of arguments, but their arguments need not be the same. `Move` statements are generated to copy any differing arguments into temporaries before the join, and the combined statement after the join will use the temporaries instead of the original arguments. Again, these `move` statements are often eliminated by the Move Eliminator.

Although more than two paths of control flow can join at the same place, the Join Merger only considers them pairwise; if more than two paths can be merged, initially only two will be merged, and the other ones will be merged in a later pass.

In order to avoid becoming stuck in an infinite loop generating `move` statements, the Fork and Join Mergers do not consider `move` statements to be similar unless both their sources and destinations are identical; under this restriction the operations of the Fork and Join Mergers must always terminate because each successful merge either reduces the number of non-`move` statements in the stmtgraph by one (and may introduce many additional `move` statements) or removes one `move` statement from the stmtgraph.

Merging similar statements instead of only identical ones is an important feature of the Fork and Join Mergers; often candidate statements differ only in that their results or arguments involve different temporaries, and the Fork and Join Mergers will merge these statements anyway, while the differing temporaries themselves will likely be merged later by the Move Eliminator. Moreover, sometimes the Join Merger becomes bolder and merges two completely different `csends` or `rsends` (as long as they have the same number of arguments). In fact, since method selectors are treated just like any other arguments, I have seen compiled code in which the Join Merger merged two `csends` or `rsends` calling different methods, a very unusual optimization indeed! In each branch just before the join, the resulting object code copied the differing method arguments into the MDP's registers and stored the appropriate method selector in a register. After the join was common code that sent the message given the method selector and arguments in the registers. Since the code to send a message is long compared to the code to load values into registers, the optimization had a net savings of five words (ten instructions) of code without significantly affecting the method's running time.

Statement Postprocessor

The Statement Postprocessor performs transformations and optimizations specific to the idiosyncrasies of the MDP architecture. These tasks were separated from the Statement Optimizer to allow the output of the Statement Optimizer to be converted back into I-Code before the Statement Postprocessor's MDP-specific transformations are done.

The Statement Postprocessor's tasks are shown in Figure 5-4 and outlined below.

Primitive Splitter

Concurrent Smalltalk defines certain associative primitive methods (`+`, `*`, `max`, `min`, `and`, `or`, `xor`, `logand`, `logor`, and `logxor`) to take an arbitrary number of arguments, while the MDP only provides instructions for operating on two arguments at a time. Therefore, each primitive that takes more than two arguments has to be converted into a sequence of primitives of two arguments. This is the function performed by the Primitive Splitter. It scans the stmtgraph and splits all primitives taking more than two arguments into sequences of shorter primitives, creating temporary variables to hold the intermediate values. The order in which the parentheses are placed in a split is not specified in Concurrent Smalltalk. Currently the compiler evaluates `(op a b c d ... z)` as if it were `(op ... (op (op (op a b) c) d) ... z)` when `op` is a primitive. This order minimizes the code size; nevertheless, the placement of parentheses can be easily changed to evaluate `op` in a tree structure if this is desirable.

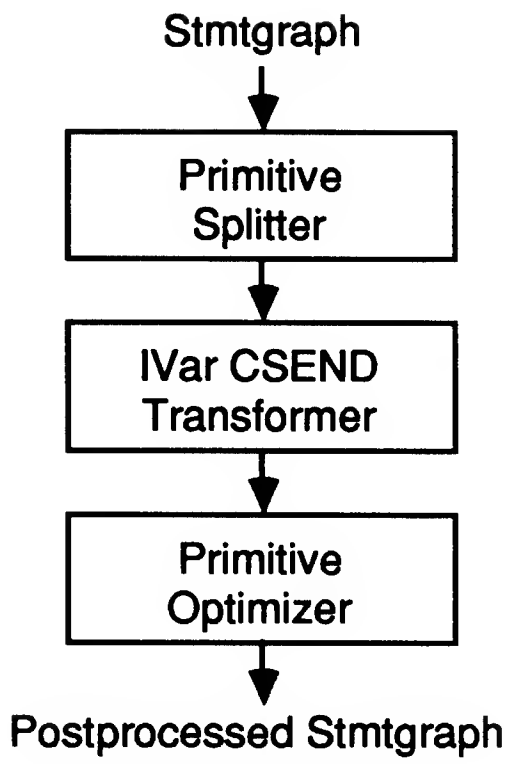


Figure 5-4. Statement Postprocessor Block Diagram.
The transformations and optimizations done by the Statement Postprocessor are mostly MDP-specific and done only once.

Instance Variable Csend Transformer

Due to a current idiosyncrasy of the operating system, the target of a `csend` can only be a local variable. The Front End and the Statement Optimizer are not aware of this restriction, however, and they generate `csend` statements with instance variables as targets. The Instance Variable `Csend` Transformer converts all such `csends` into `csends` with temporary local variables as targets followed by `moves` into appropriate instance variables.

Primitive Optimizer

The Primitive Optimizer performs one final pass at optimization of primitives. It differs from the Constant Folder in that it performs MDP-specific optimizations. The two optimizations it currently performs are converting multiplications and divisions by powers of two into arithmetic shifts.

Statement Printer

A routine, output-stmtgraph, is provided to print a stmtgraph in an I-Code-like format. The routine is mainly for debugging purposes, but it can also be used to convert a stmtgraph back into a variant of I-Code. Output-stmtgraph prints only the operation, target, method, and arguments of the stmts, ignoring the other fields used by the Statement Analyzer and Optimizer. Output-stmtgraph contains an algorithm similar to that of the Assembly Code Generator's Branch Inserter for inserting unconditional branches and labels in before statements that are targets of branches into the printed I-Code. This routine produced some of the listings in Chapter 8.

Chapter 6. Instruction Generator

The Instruction Generator converts the stmtgraph into a module, which is an ordered digraph of MDP assembly language instructions. In order to perform this function, the Instruction Generator has to find an appropriate order for the statements in the stmtgraph as well as allocate the local variables in the stmtgraph into either registers or memory locations. These functions are performed by the Linearizer and the Variable Allocator. The Variable Allocator creates a varinfo record that describes the final assignments of variables to locations as well as some statistics about the stmtgraph. After the order of statements and the locations of the variables are known, the actual generation of instructions (insts) can begin. The Stmt Compiler uses the Frame Handler to keep track of the data in the registers and memory locations while it is generating the instructions.

Insts

```
(defstruct (inst (:include dinode) (:print-function print-inst))
  label     ;The label number for this instruction.
  op
  src1
  src2
  dst
  reads     ;Map of registers whose values are used by this instruction.
  writes    ;Map of registers written or trashed by this instruction.
  live      ;Map of registers live at the end of this instruction.
  vlive     ;Map of vlocs live at the end of this instruction.
  pc        ;The program counter in half-words.
  next      ;The next instruction in the output code or NIL if there is none.
  prev)     ;The previous instruction in the output code or NIL if there is none.
```

An inst record describes an MDP assembly language instruction. The instruction has an operand op, up to two source locations src1 and src2, and a destination location dst. These fields are enough to completely describe the instruction. Please refer to Appendix A for details on the MDP instruction set. The remaining fields contain additional data about the instruction such as the registers read and written by the instruction, the context variables live at the end of the instruction, the address of the instruction, and links to the previous and next instruction in the method code (these links are static code location links as opposed to the flow of control links that are edges of the stmt and inst digraphs).

The possible locations that may be used as the src1, src2, or dst fields of an instruction are listed in Table 6-1.

Table 6-1. Location Syntax

`(sconst` <constant>`)`	Short constant (one that can be generated by an MDP addressing mode).
`(lconst` <constant>`)`	Long constant (one that requires a `DC` instruction).
`(reg` <number>`)`	MDP data register `R0`, `R1`, `R2`, or `R3`.
`(areg` <number>`)`	MDP address register `A0`, `A1`, `A2`, or `A3`.
`(sreg` <name>`)`	MDP special register <name>.
`(vloc` <number>`)`	Context variable at offset <number>.
`(iloc` <number>`)`	Instance variable at offset <number> in the instance object.
`(aloc` <number>`)`	Argument at offset <number> in the message.
`(rel)`	Filler for branch addressing mode (see Assembly Code Generator).

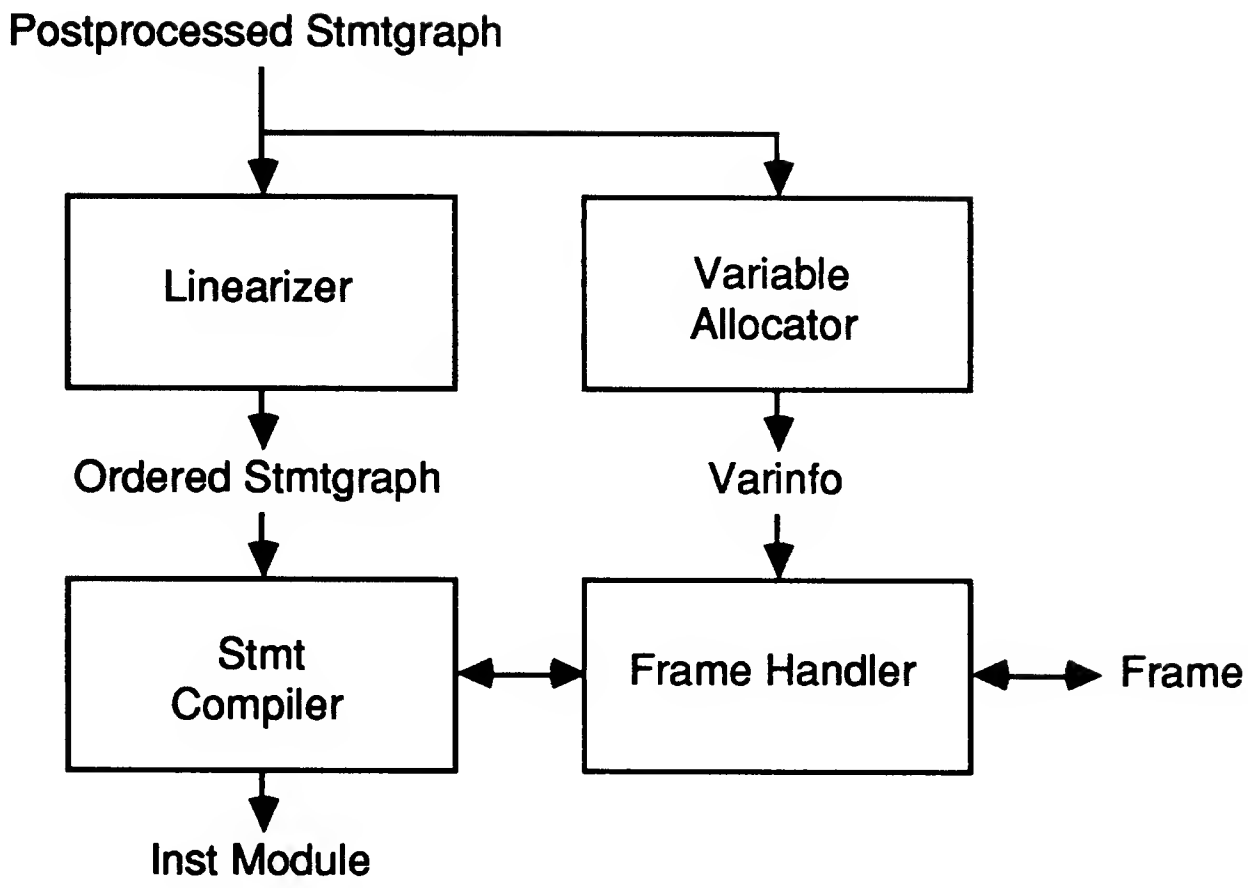


Figure 6-1. Instruction Generator Block Diagram.

Linearizer

The Linearizer calls the linearize function (see Appendix C) of the digraph Utilities to produce an ordering of the stmtgraph's nodes that tries to minimize the number and total length of branches in the stmtgraph. The statements are compiled into insts in that order, so the static links (the prev and next fields of insts) between the instructions in the module will reflect the order on statements in the stmtgraph defined by linearize.

Variable Allocator

The Variable Allocator calculates some statistics about the stmtgraph and assigns all local variables that are used into either registers or context locations. The assignment process proceeds by first finding all local variables that are actually referenced in the stmtgraph. Due to the statement optimizations such as the Move Eliminator, many local variables are actually never referenced. The referenced-vars function returns a bmap of all variables that are used in the stmtgraph.

The bmap of referenced variables is then passed to the Register Allocator which attempts to place as many variables as possible into registers. It reports which variables it was able to put into registers; the remaining ones are passed to the Context Variable Allocator, which packs them as tightly as it can into context slots. The outputs of both Allocators are stored into a varinfo record that lists the location of each local variable and whether a context is necessary or not.

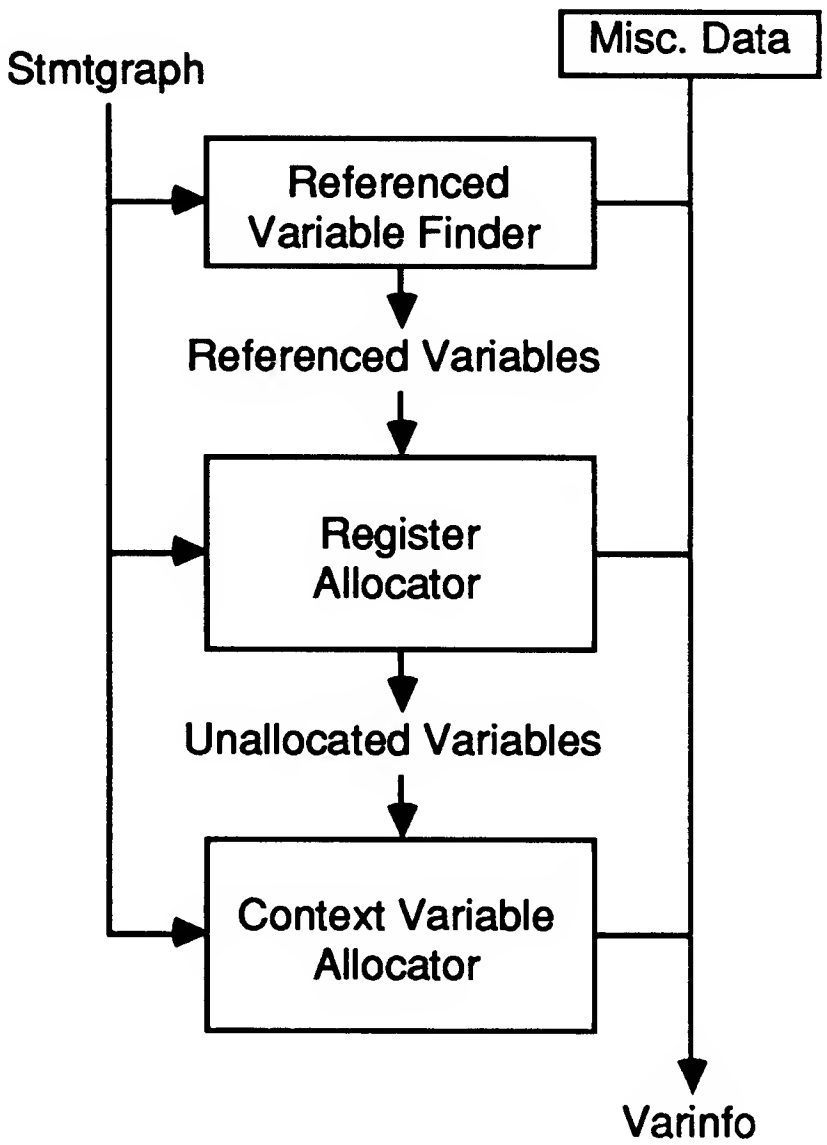


Figure 6-2. Variable Allocator Block Diagram.

Register Allocator

The Register Allocator tries to allocate local variables in the MDP's registers. Since variables that are targets of `csends` must be in memory, they are ineligible for register allocation and immediately disqualified. The other variables are either always in the context in memory or always in a register. No attempt is made to keep a variable in a register for a portion of its lifetime and in memory for the rest of its lifetime, although some of the frame optimizations done by the Stmt Compiler and Frame Handler may have this effect.

The variables eligible for register allocation are prioritized according to the formula

$$\text{Priority} = \frac{\text{nrefs}}{\max(\text{nrefs},\text{nlive})}$$

where nrefs is the number of references to the variable present in the stmtgraph and nlive is the number of statements during which the variable is live. The highest priority variables are considered first. The effect of this system of priorities is to make variables that are used often and have short lifetimes be more likely to be allocated in registers than variables that are used rarely and have long lifetimes. The former variables use little time in registers, so considering them first greatly increases the number of variables that will fit in the registers.

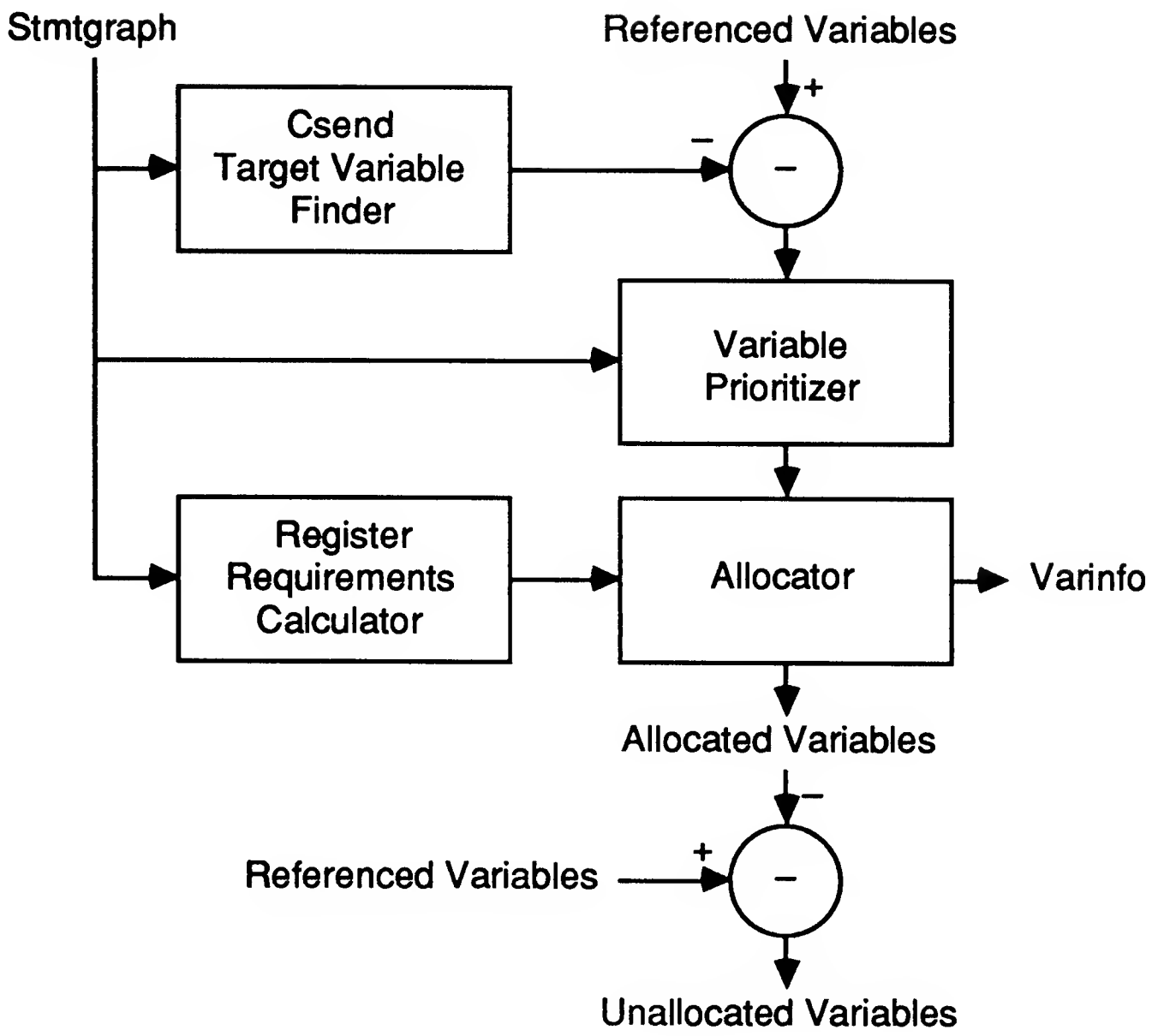


Figure 6-3. Register Allocator Block Diagram.

The Register Allocator needs to know how many registers it can allocate. The Stmt Compiler needs temporary registers to compile some statements, so the Register Allocator contains a function, the Register Requirements Calculator, that yields an estimate of how many temporary registers are required to compile each statement in the stmtgraph. Since compiling some statements requires the use of registers with specific numbers (for example, some statements make system calls that require an argument in `R1`), each estimate includes both the maximum number of temporary registers needed to compile the statement and the specific register numbers to be allocated for the statement. The estimates are always conservative to prevent the Stmt Compiler from running out of registers, as once a variable has been assigned to a register, there is no way to undo that assignment.

Once the variables have been prioritized and the amount of space available in the registers is known, the assignment process begins. A greedy algorithm is used. The variables are considered in order of decreasing priority. For each variable, the Allocator considers each statement in which the variable is live. If there is a common free register in all such statements, the variable is assigned to that register, and that register is marked as busy. Regardless of whether the variable was assigned or not, the variable with the next lowest priority is considered until all variables have been considered. The variables which were not allocated to registers are then passed to the Context Variable Allocator.

In practice, despite the low number of registers on the MDP, the Register Allocator is able to allocate almost all variables to the registers. The vast majority of variables are temporaries

with one-instruction lifetimes, giving them maximum priority, and all such variables can be allocated to registers. Only a few remain to be processed by the Context Variable Allocator.

Context Variable Allocator

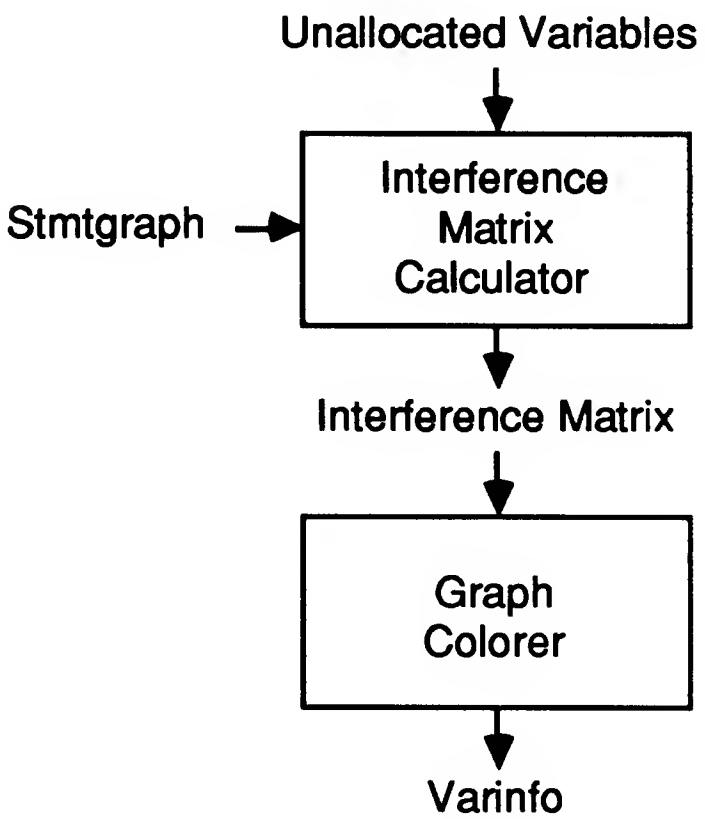


Figure 6-4. Context Variable Allocator Block Diagram.

The Context Variable Allocator is a procedure in the Optimist compiler that is not often found in other compilers. The Context Variable Allocator's goal is packing the few remaining unallocated variables into as few context slots as possible. Context space is very limited because the MDP is only capable of directly addressing the first sixteen words of a context and the operating system reserves five of them, leaving only eleven for storage of local variables; thus, context local variables must be conserved.

The Context Variable Allocator works by calculating an *interference matrix* of the unallocated variables. The interference matrix is a two-dimensional Boolean matrix that indicates whether any two given variables are ever simultaneously live at any point in the stmtgraph. If so, then the variables interfere and cannot be assigned to the same context location.

The interference matrix is passed to a general graph coloring algorithm that tries to color the graph represented by the matrix (each variable is a vertex, and two variables are connected by an edge iff they interfere) with as few colors as possible (the colors represent context memory locations) so that no two vertices with the same color are connected by an edge. In general this problem is NP-complete, but a good heuristic for solving it exists [4].

The main insight is to note that if an n-coloring of the graph exists and the graph contains a vertex A with degree less than n, then A can be removed from the graph, the new graph n-colored, and then A assigned a color different from any of its (at most n-1) neighbors. Since n is initially not known, the Optimist's coloring algorithm assumes that n is 1 until it can remove no more vertices with degree 0 and still has a nonempty graph left. At that point it revises its estimate of n to 2 and proceeds to remove all vertices with degrees 0 and 1. Note that n is often less than the maximum degree of a vertex in the original graph because removing vertices often lowers other vertices' degrees.

One final improvement in the coloring algorithm exists in the phase in which vertices are assigned colors after having been removed from the graph. At that point the algorithm tries to avoid new colors as long as possible, preferring to color vertices that are being put back into the graph with colors that were already used. Sometimes this will yield a better coloring of the graph than n would indicate. The simplest example of this phenomenon is the graph composed of the four vertices of a square—the algorithm will able to 2-color the graph even though it will reach an estimate n of 3.

In all of the practical cases that I have encountered so far, the Context Variable Allocator always yielded an optimal allocation of variables to context slots.

Stmt Compiler

The Stmt Compiler compiles each stmt in the stmtgraph into a number of MDP instructions. The Utilities function map-digraph is used to construct the module. While the Stmt Compiler is compiling each statement, the Frame Handler keeps track of the state of the variables.

The Stmt Compiler is too long to describe in detail here; instead, only the highlights will be presented below. Please see Appendix D for the details about the implementation of the Stmt Compiler.

Frame Handler

The Frame Handler works with a data structure called a frame:

```
(defstruct (frame (:copier copy-frame1))
  varinfo                       ;Global varinfo assignments.
  (regs (make-array '(4)))      ;Array of known register slot values.
  (lockmap b0 :type bmap)       ;Bmap of register locks.
  (waiting b0 :type bmap)       ;Bmap of unforced slots.
  (migrate t)                   ;True if the instance object could have migrated away.
  (lru-regs '(0 1 2 3)))        ;List of registers in order from most to least recently used.
```

The frame contains the entire state of the variables at some place in the stmtgraph. The Stmt Compiler is able to interrogate the frame about the location of a specific local variable, whether a variable's value is available, and whether the instance object could have migrated away to another node (which it can whenever there is an opportunity to suspend execution of the method). The Frame Handler knows about the Variable Allocator's assignments through the varinfo record. In addition, it also keeps track of the current values in the registers; if the Stmt Compiler requests an access to a context variable but whose value just happens to be present in a register, the Frame Handler will return the register to the Stmt Compiler. The Frame Handler also is in charge of allocating free registers for temporary use by the Stmt Compiler; it uses the least recently used strategy to allocate these temporaries and avoids allocating registers that contain variables. Finally, the Stmt Compiler can ask the Frame Handler to lock a certain register, preventing it from being allocated, and the Frame Handler will honor that request.

The frame contains a significant amount of data outlined above in addition the varinfo record. Since the instructions are usually generated in the order in which they are executed, such data can be maintained and be useful. Problems do arise, though, when a join is encountered in the stmtgraph. In that case the Frame Handler compares the two or more frames in the joining paths and picks the most conservative frame out of the two or more—i.e. if the frames disagree about what is in register R1, the resulting frame will contain no information about the contents of R1. Also, if one of the frames of the joining paths is unavailable (because, say, that section of code has not yet been compiled due to a loop), the Frame Handler selects the most conservative frame possible which contains only the information from the varinfo record.

Issues in Compiling Statements

The Stmt Compiler, for its part, tries to be as cooperative with the Frame Handler as possible. It lets the Frame Handler examine every instruction that it generates so that the Frame Handler always has a current idea of what values are in what registers; if the Stmt Compiler ever neglected to tell the Frame Handler about a change to a register, the frame would become obsolete with potentially disastrous results. The process of updating the frame has been made very simple and mechanical to try to avoid this kind of error.

The Stmt Compiler also helps the Frame Handler by pointing out possible aliases in the code it generates. For example, when the Stmt Compiler outputs a `move` instruction, it informs the Frame Handler that the source and destination locations are temporarily aliases of each other until one of them is changed. The Frame Handler keeps track of such aliases, and, if a location is requested but its alias can be accessed easier, it will return the alias.

Uninterruptibility of Sends

The Stmt Compiler uses the Frame Handler's special services for compiling some statements. A major issue in designing the Stmt Compiler was preventing faults in the middle of `csend`, `rsend`, and `reply` statements. Due to the design of the MDP, once the first word of a message has been sent onto the network by one of these statements, sending must continue uninterrupted until the entire message has been sent. A fault caused by accessing a context variable with an unavailable value would crash the system. An access of the instance object when it has migrated away to another node would have similar consequences. To avoid these difficulties, the Stmt Compiler checks each slot that is going to be sent in `csend`, `rsend`, and `reply` statements. If that slot is not guaranteed to be available, the Stmt Compiler issues a statement to touch that slot before sending begins. Since most of the time values can be shown to be available, having the Frame Handler keep track of the availability data saves a lot of unnecessary code.

Preventing Limbo Variables

There is another unobvious issue in the design of the Stmt Compiler. Consider the code in Figure 6-5a. This pair of `csend` statements illustrates a problem that could arise in the code generated by the compiler if the compiler were not very careful. When two consecutive stores are made to a variable, the first of which makes the variable's value unavailable, the variable enters the *limbo* state. There is no way to tell when the value of a limbo variable might change, and that variable is for all practical purposes useless from that point on. The compiler does not let variables enter the limbo state by touching a variable before a store to it whenever the variable's value is not guaranteed to be available.

Deallocating Variables

On a similar note, the compiler issues code to touch all context variables whose values are not guaranteed to be available just before the context is deallocated. This forces the method to wait until all replies come back before the context can be freed. Without this precaution, a reply could come back after the context has been deallocated and reused for another method, clobbering another method's variables.

The `touch` operations are done after the method has sent its reply to its caller, so even if a touch causes a wait, that wait does not slow down the program that is running because the reply has already been sent to the caller. The only externally visible effect is that the context remains allocated for a somewhat longer time than it would otherwise have been.

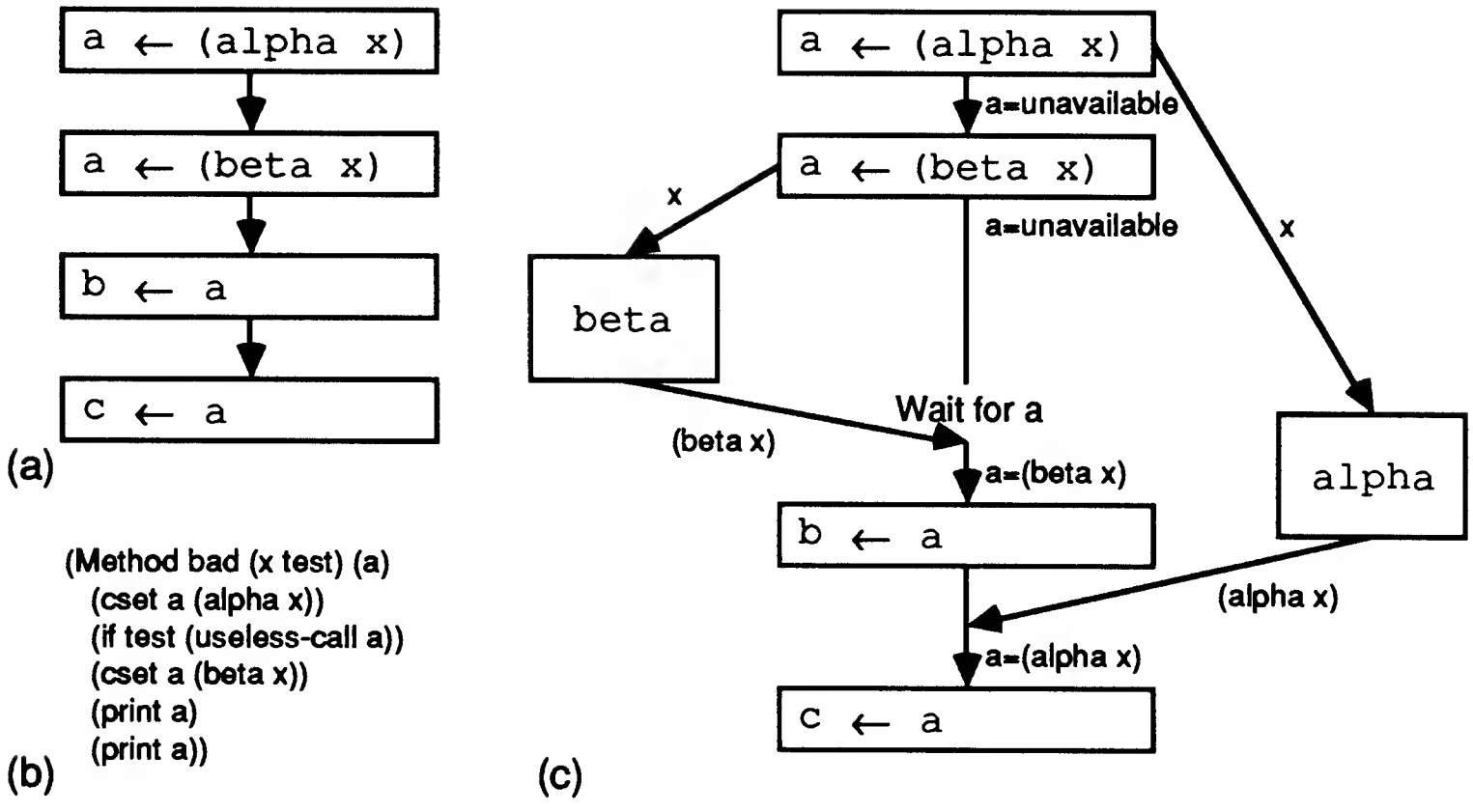


Figure 6-5. The Peril of Limbo Variables.

A variable enters the limbo state when two `csends` (or a `csend` followed by any store) are made to it without touching it in between. Sample code that might cause the situation is shown in (a). Actually, due to the optimizations in the Statement Optimizer, the first `csend` in (a) would be optimized to a `nil` target, eliminating the problem, but the problem could still be made to appear in a more complicated example like (b) where `test` is always false but the compiler does not know that.

The problem with a variable in the limbo state is that its value might change at any time without warning. (c) shows what might happens when a variable `a` is in limbo. After both `csends` are sent, the value of `a` is `unavailable`. The `move` from `a` to `b` correctly waits until the value of `a` is available; let's say that `beta` replies first, so the value of `a` becomes `(beta x)`. The `move` then proceeds, and everything is fine until the reply from `alpha` comes back, at which point it clobbers the value of `a` without any warning. When `a` is in limbo, there is no way to tell whether `alpha` (or `beta`) has returned its value or not.

The only good way of dealing with limbo variables is to make sure that they don't arise. An apparent alternative, checking whether the value of the variable is unavailable at the time a called method responds, will not work. In the above example the compiler would touch `a` between the two `csends` to make sure it does not enter the limbo state.

Context Optimization

When there are no context variables, the Stmt Compiler does not compile the operating system calls to allocate a context at the beginning of the method and deallocate it at the end, resulting in a significant speed improvement for the method as well as a small (three instruction) space improvement. Similarly, if there are no references to the instance object in the method, then no code is generated to get the address of the instance object into register `A2`, resulting in a small speed increase and space saving.

Chapter 7. Assembly Code Generator

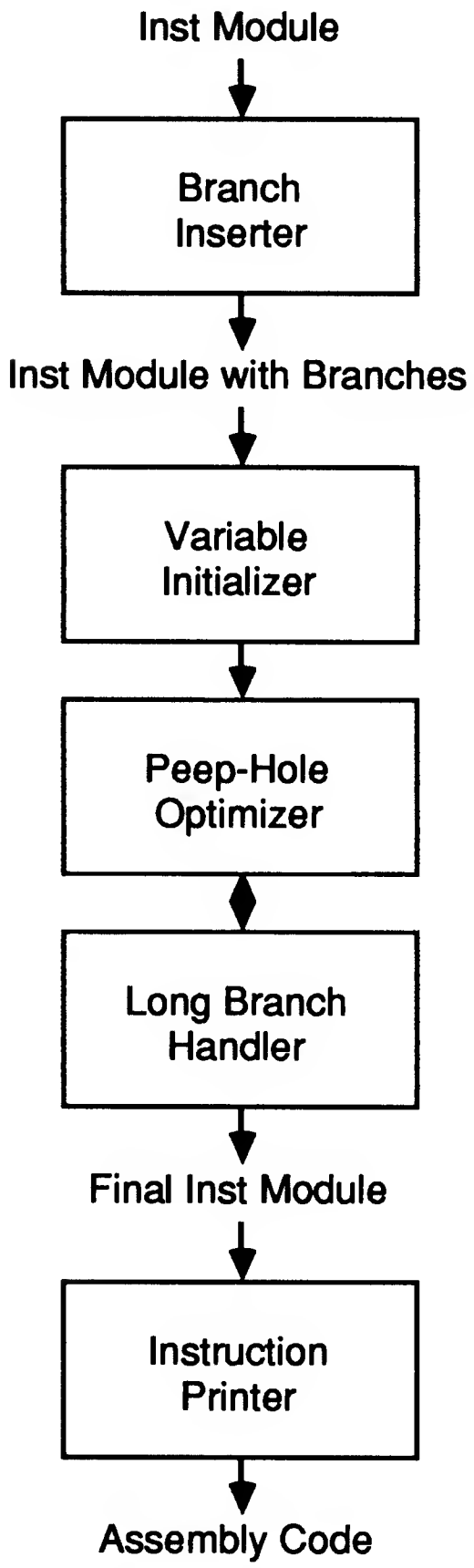


Figure 7-1. Assembly Code Generator Diagram.

The Assembly Code Generator performs transformations and optimizations on the module created by the Instruction Generator. Since the Instruction Generator relied on the module topology to indicate paths of control flow, the Branch Inserter has to insert branches into the module. The Variable Initializer initializes selected context variables where appropriate. The Peep-Hole Optimizer performs several instruction optimizations such as shifting instructions to align DC instructions to word boundaries and combining SEND and SENDE instructions into

SEND2 and SEND2E. In conjunction with the Peep-Hole Optimizer, the Long Branch Handler extends short branches that do not reach their destinations into long branches, which is a nontrivial operation on the MDP because long branches require the use of register R0, while short branches do not. Finally, the Instruction Printer outputs the module into a file as a series of assembly language statements.

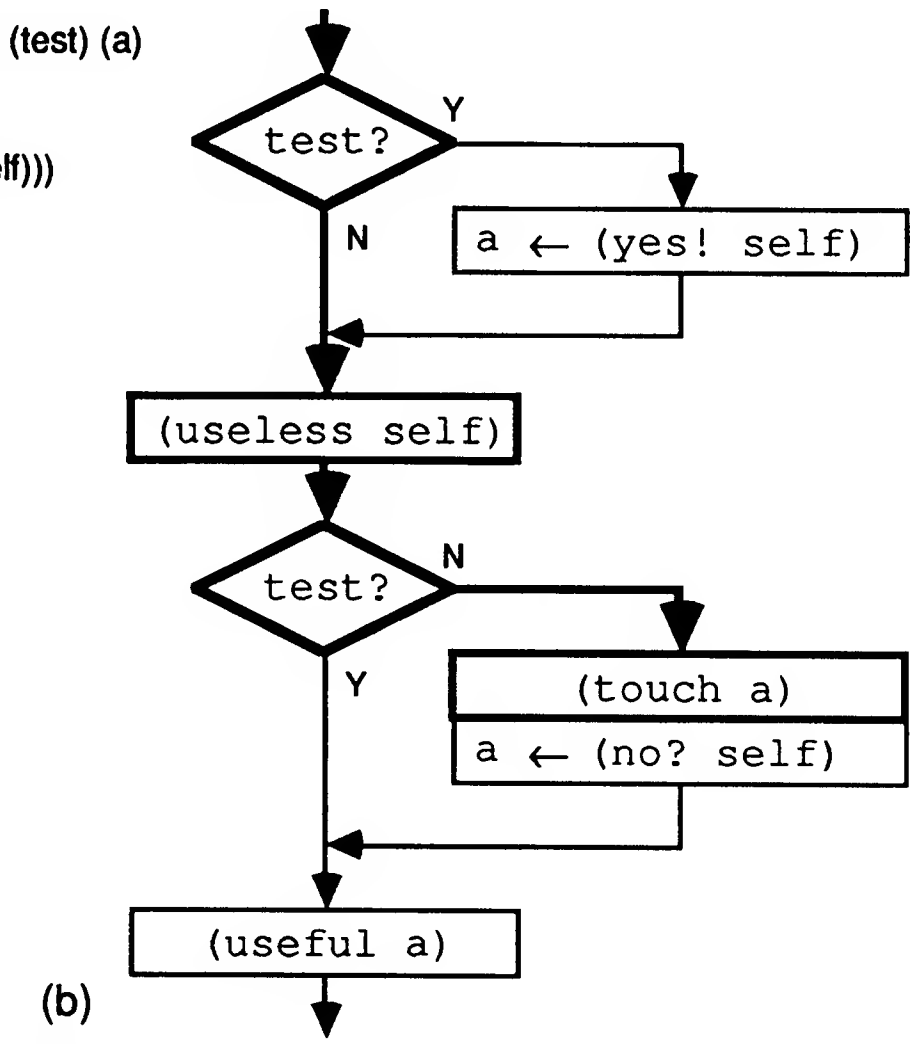
Branch Inserter

The Branch Inserter scans the stmtgraph and finds all places where control flow does not simply pass from one instruction to the next. It inserts unconditional branches in all places where the successor of an instruction according to the digraph is not the next instruction in the static sequence. Conditions are also considered; if one successor of a condition is the next instruction in the static sequence, the condition is made into a conditional branch (possibly reversing the condition). If neither successor is the next instruction in the static sequence, a conditional branch is made, followed by an unconditional branch.

Variable Initializer

```
(Method Integer Init-Sample (test) (a)
  (if test (cset a (yes! self)))
  (useless self)
  (if (not test) (cset a (no! self)))
  (useful a))
```

(a)



(b)

Figure 7-2. The Need for the Variable Initializer.
The method in (a) compiles to the code sketched in (b). The Stmt Compiler inserts the touch instruction before the second cset to a to avoid the possibility of a going into the limbo state (see Figure 6-5); the Stmt Compiler does not realize that the two csets to a could never both be executed (and even if it did, other pathological examples could be constructed).

Thus, when test is false, an uninitialized a would be referenced by the touch instruction, even though the source program never references an uninitialized variable. Since referencing uninitialized memory is dangerous (because that memory may contain the unavailable value, causing the method to wait forever), the compiler is under an obligation to initialize a.

The Variable Initializer inserts code to store NIL in any context variables that are live at the beginning of the method. The Variable Initializer is not intended to provide a default initial value for the method's local variables; in fact, it does not initialize register variables. Instead, the Variable Initializer is necessary to handle some pathological cases in which the extra touches put in by the Stmt Compiler to prevent Limbo Variables can refer to uninitialized variables. Touching an uninitialized variable is dangerous because that variable just might happen to have the unavailable value, causing the method to hang. See Figure 7-2 for an example in which the source code does not refer to any uninitialized variables yet in which an uninitialized variable reference is created by the Limbo Variable elimination process.

Peep-Hole Optimizer

The Peep-Hole Optimizer currently performs two transformations: combining SENDs and aligning DCs. In addition, it includes the PC Scanner, a routine that finds the address of each instruction in the module. The DC Aligner is actually combined with the PC Scanner to improve efficiency; this combination is not essential to the algorithms, though, and the two functions will be explained separately.

SEND Combiner

The Instruction Optimizer first tries to combine SEND and SENDE instructions into SEND2s and SEND2Es, which send two values instead of just one (see Appendix A). It scans the module and considers every SEND and SENDE instruction. It scans backwards from that instruction until the beginning of its basic block for another SEND instruction. If it finds one, it checks whether the instructions between the SEND instruction and the SEND or SENDE can all be moved either before the leading SEND or after the trailing SEND or SENDE. It uses a utility subroutine, insts-commute?, to test whether one instruction can be moved past another without affecting the semantics of the program. Insts-commute? considers such factors as whether one instruction changes a register used by the other, whether one can change the flow of control, and whether they both use the same resource such as the network or the stack. If all instructions between the SEND and the SEND or SENDE can be moved out, all of these instructions are, in fact, moved out of that interval, and the SEND is combined with the SEND or SENDE to make a SEND2 or SEND2E instruction. The process continues until no more such combinations can be made.

DC Aligner

DC instructions are constants embedded in the method code. When the MDP attempts to execute a constant, it just loads it into register R0 and proceeds with the next instruction. Normally two instructions can fit into a word, but constants must be word-aligned, which forces the MDPSim assembler to issue a no-operation instruction if the PC was not word-aligned. The objective of the DC Aligner is to try to align as many DCs as possible to word boundaries to prevent wasted code and time.

The DC Aligner looks for DC instructions at mid-word addresses. If it finds one, it tries to shift it to a word boundary by exchanging it with either the previous or the next instruction using insts-commute? to test whether such an exchange would be legal. If the exchange can be done, it is done. Otherwise, the DC instruction is left as is; the assembler will automatically align the DC.

PC Scanner

The PC Scanner scans through the module advancing the PC by one instruction (1/2 word) for each instruction except DCs which are one word (two instructions) long. It aligns the PC to a

word boundary at every DC and every instruction that is a destination of a branch (branches can only branch to word boundaries).

Long Branch Handler

The Long Branch Handler is the most complicated routine in the Assembly Code Generator. It scans the module and checks every short branch it finds to make sure that it can reach its destination within the limited MDP branching range (-15 to +16 words). It uses several heuristics to try to extend short branches that do not reach their destinations into long branches.

The first heuristic tried is branch chaining (Figure 7-3a) [13]. When considering a short branch that does not reach the destination, the Long Branch Handler checks whether there is any other branch instruction in the module that branches to the same destination, and, if so, whether that branch is within the short branching range of the current location. Such a check can easily be done by checking the destination instruction's predecessors in the inst digraph. If the check succeeds, the destination of the branch is changed to point to the other branch that does reach the destination.

If the above heuristic fails and the branch is an unconditional one, the branch is changed to a DC / BR R0 instruction sequence (Figure 7-3b). This instruction sequence uses the R0 register, while the short branch does not; thus, in order to avoid generating bad code, the Long Branch Handler checks that register R0 is not live at the point of the branch instruction. If R0 is live, the Long Branch Handler gives up and signals an error. It is the Instruction Generator's duty to make sure that R0 is not live at any point at which an unconditional long branch could appear; this is why the linearization is done before the Stmt Compiler compiles the statements into instructions.

If the branch is conditional, the Long Branch Handler checks whether it is followed by a short unconditional branch. If so, the condition is reversed and the two destination addresses interchanged, reducing the problem to extending an unconditional branch (Figure 7-3c). This situation is handled as above. There is a possibility that interchanging the two destination addresses overflows the range of the short branch. If this happens, the overflow will be handled on the next pass of the Long Branch Handler.

If the branch is conditional and not followed by an unconditional short branch, the source register for the condition is considered. If it is not R0, the branch is changed into a DC / BR Rn,R0 instruction sequence in a similar manner as above (Figure 7-3d). If the condition register is R0, the Long Branch Handler attempts to move the condition to some other register that is not live. If it is successful, the case is handled as before (Figure 7-3e). If not, the branch is changed to a short branch around a long unconditional branch instruction (Figure 7-3f).

In several of the above cases an extra DC instruction is introduced into the module. The DC Aligner is run after the Long Branch Extender to try to align these DCs to word boundaries. The DC Aligner and the Long Branch Extender are run repeatedly until neither makes any changes to the module. It can be shown that these two handlers will not continue to modify the module indefinitely.

Instruction Printer

The Instruction Printer is the final stage of the Optimist compiler. It prints the MDP assembly language instructions in the module onto a stream. It outputs the entire module as a CODE message, the format of which is defined in Appendix B. Before outputting the

module, it scans the entire module and places labels on statements that are destinations of branches; the branches then refer to these labels instead of less readable numeric offsets. Please refer to [10] and [9] for the complete syntax of the assembly language. Chapter 8 contains sample outputs of the Instruction Printer.

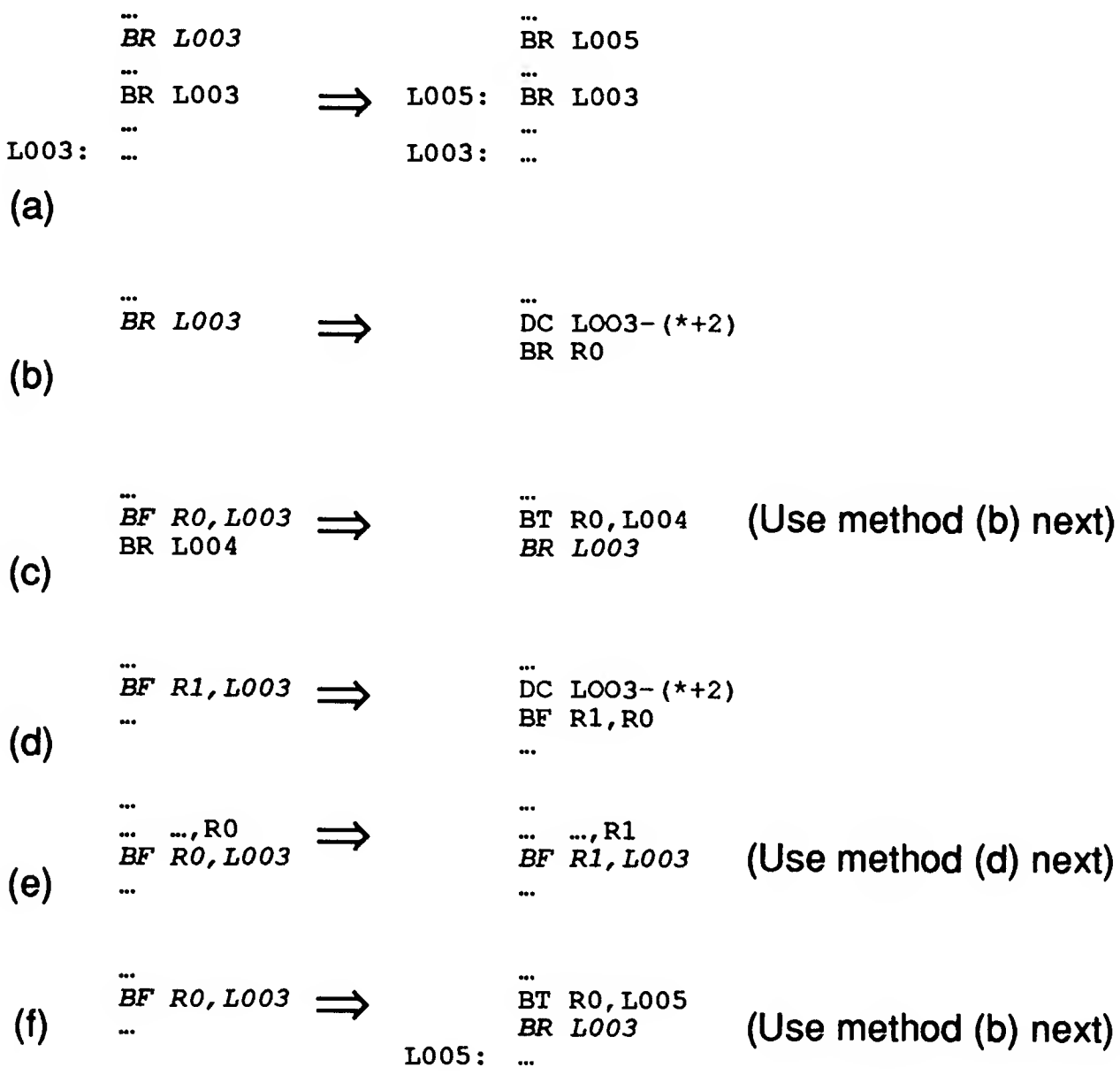


Figure 7-3. Expanding Branches.

There are seven methods of extending short branches into long ones. These methods are outlined above. In each situation the overflowing short branch is in italics. The methods are branch chaining (a), simple-extending an unconditional branch (b), reversing a conditional branch (c), simple-extending a conditional branch (d), changing the condition register (e), and branching around an unconditional branch (f).

Chapter 8. Examples

Guided Tour

Below is an example of a typical small Concurrent Smalltalk method and the stages of its compilations. The example was intended to be simple, illustrative, useful, and typical; it was not contrived to exhibit the compiler's best performance, nor does it invoke all of the compiler's optimizations.

The source Concurrent Smalltalk method is shown in Figure 8-1. The method belongs to the class pair, also illustrated in Figure 8-1.

```
(class pair (object) car cdr)

(method pair length1 (n) ()
      (if (eq cdr 'nil)
        (+ 1 n)
        (length1 cdr (+ 1 n))))
```

Figure 8-1. Sample Concurrent Smalltalk Method.
The method returns n plus the length of a Lisp-like list defined by objects called pairs that have two fields: car, the datum field; and cdr, a pointer to the next object in the list.

The Front End converts the source Concurrent Smalltalk method into I-Code shown in Figure 8-2. The I-Code Preprocessor then processes the I-Code to add `enter` and `exit` statements and merge all `return` statements into one `reply` at the end (Figure 8-3).

```
(CSEND (TEMP 0) (METHOD EQ) (IVAR 1) (CONST NIL))
(FALSEJUMP (TEMP 0) 0)
(CSEND (TEMP 1) (METHOD +) (CONST 1) (ARG 0))
(JUMP 1)
(LABEL 0)
(CSEND (TEMP 2) (METHOD +) (CONST 1) (ARG 0))
(CSEND (TEMP 1) (METHOD LENGTH1) (IVAR 1) (TEMP 2))
(LABEL 1)
(RETURN-X (TEMP 1))
```

Figure 8-2. I-Code for the Sample Method.
The I-Code output by the Front End is a literal translation of the source code with few optimizations. At this point all method calls, including primitives, are compiled as `csends`. The `return-x` statement at the end performs the same role as a `return` statement—it returns the value of the method to the caller.

```
(ENTER)
(CSEND (TEMP 0) (METHOD EQ) (IVAR 1) (CONST NIL))
(FALSEJUMP (TEMP 0) 0)
(CSEND (TEMP 1) (METHOD +) (CONST 1) (ARG 0))
(JUMP 1)
(LABEL 0)
(CSEND (TEMP 2) (METHOD +) (CONST 1) (ARG 0))
(CSEND (TEMP 1) (METHOD LENGTH1) (IVAR 1) (TEMP 2))
(LABEL 1)
(MOVE (TEMP RETURN-VALUE) (TEMP 1))
(JUMP RETURN)
(JUMP EXIT)
(LABEL RETURN)
(REPLY (TEMP RETURN-VALUE))
(LABEL EXIT)
(EXIT)
```

Figure 8-3. Processed I-Code for the Sample Method.
The preprocessor added some dead code and extra branches in its efforts to merge `return` and `exit` statements. That dead code will be removed when the I-Code is diagraphized.

At this point the Diagraphizer and Canonarizer are invoked to produce a stmtgraph of the I-Code. The printout of an entire stmtgraph is too long and complicated; instead, the result of output-stmtgraph run on the stmtgraph is shown in Figure 8-4. It should be kept in mind that output-stmtgraph inserts labels and unconditional branches in its output that are not present in the stmtgraph.

```
(ENTER)
(PRIMITIVE (VAR . 0) EQ (IVAR . 1) (CONST 0 . 0))
(CONDITION BF (VAR . 0) 193)
(PRIMITIVE (VAR . 1) + (CONST 1 . 1) (ARG . 0))
(LABEL 195)
(MOVE (VAR . 2) (VAR . 1))
(REPLY (VAR . 2))
(EXIT)
(LABEL 193)
(PRIMITIVE (VAR . 3) + (CONST 1 . 1) (ARG . 0))
(CSEND (VAR . 1) (CONST METHOD . LENGTH1) (IVAR . 1) (VAR . 3))
(JUMP 195)
```

Figure 8-4. Initial Stmtgraph of the Sample Method.
The primitives have been recognized, constants reformatted (`(const a . b)` indicates a MDP word with tag `a` and data `b`), and local variables renumbered to start at 0.

Next the Dead Definition Eliminator is invoked without any effects. It is followed by the Move Eliminator, which does successfully remove the unnecessary `move` statement.

```
(ENTER)
(PRIMITIVE (VAR . 0) EQ (IVAR . 1) (CONST 0 . 0))
(CONDITION BF (VAR . 0) 193)
(PRIMITIVE (VAR . 1) + (CONST 1 . 1) (ARG . 0))
(LABEL 198)
(REPLY (VAR . 1))
(EXIT)
(LABEL 193)
(PRIMITIVE (VAR . 3) + (CONST 1 . 1) (ARG . 0))
(CSEND (VAR . 1) (CONST METHOD . LENGTH1) (IVAR . 1) (VAR . 3))
(JUMP 198)
```

Figure 8-5. Stmtgraph with `Move` Statement Removed.
The Move Eliminator discovered and removed the unnecessary `move` statement.

There are no `touch` statements, so the Touch Eliminator does nothing. However, the Dataflow Optimizer spots that variable 0 is the result of testing ivar1 against `nil`, and it converts the branch-if-false into a branch-if-not-`nil` that tests ivar1 directly.

```
(ENTER)
(PRIMITIVE (VAR . 0) EQ (IVAR . 1) (CONST 0 . 0))
(CONDITION BNNIL (IVAR . 1) 193)
(PRIMITIVE (VAR . 1) + (CONST 1 . 1) (ARG . 0))
(LABEL 198)
(REPLY (VAR . 1))
(EXIT)
(LABEL 193)
(PRIMITIVE (VAR . 3) + (CONST 1 . 1) (ARG . 0))
(CSEND (VAR . 1) (CONST METHOD . LENGTH1) (IVAR . 1) (VAR . 3))
(JUMP 198)
```

Figure 8-6. Stmtgraph Processed by Dataflow Optimizer.

The Dataflow Optimizer optimized the conditional branch `bf` to a `bnnil`, but it did not remove the now-redundant original test of ivar1 against `nil`.

The Constant Folder does not find any constant expressions it can remove. The Tail Forwarder does spot, however, that the `csend` is followed by the `reply` statement.

```
(ENTER)
(PRIMITIVE (VAR . 0) EQ (IVAR . 1) (CONST 0 . 0))
(CONDITION BNNIL (IVAR . 1) 193)
(PRIMITIVE (VAR . 1) + (CONST 1 . 1) (ARG . 0))
(REPLY (VAR . 1))
(LABEL 199)
(EXIT)
(LABEL 193)
(PRIMITIVE (VAR . 3) + (CONST 1 . 1) (ARG . 0))
(RSEND (CONST METHOD . LENGTH1) (IVAR . 1) (VAR . 3))
(JUMP 199)
```

Figure 8-7. Stmtgraph after Tail Forwarding.

The `csend` has been converted to an `rsend` followed by a branch around the `reply`. This optimization not only decreases the code size, but it will also eliminate the need to allocate a context in Figure 8-13.

The Conditional Folder does not find any `condition` both of whose branches point to the same place, and the Join Merger is unsuccessful. Nevertheless, the Fork Merger does find the + primitive on both sides of the conditional, and it moves it before the conditional.

```
(ENTER)
(PRIMITIVE (VAR . 0) EQ (IVAR . 1) (CONST 0 . 0))
(MOVE (VAR . 5) (IVAR . 1))
(PRIMITIVE (VAR . 4) + (CONST 1 . 1) (ARG . 0))
(CONDITION BNNIL (VAR . 5) 201)
(MOVE (VAR . 1) (VAR . 4))
(REPLY (VAR . 1))
(LABEL 199)
(EXIT)
(LABEL 201)
(MOVE (VAR . 3) (VAR . 4))
(RSEND (CONST METHOD . LENGTH1) (IVAR . 1) (VAR . 3))
(JUMP 199)
```

Figure 8-8. Stmtgraph after First Optimization Pass.

The Fork Merger successfully moved the + primitive before the bnnil, eliminating one + but also adding three `move` statements.

Since the above optimizations did change the stmtgraph, the Statement Optimizer makes another pass through the optimizations. The Dead Definition Eliminator is called again, and this time it removes the redundant eq left over by the Dataflow Optimizer from the last pass.

```
(ENTER)
(MOVE (VAR . 5) (IVAR . 1))
(PRIMITIVE (VAR . 4) + (CONST 1 . 1) (ARG . 0))
(CONDITION BNNIL (VAR . 5) 201)
(MOVE (VAR . 1) (VAR . 4))
(REPLY (VAR . 1))
(LABEL 199)
(EXIT)
(LABEL 201)
(MOVE (VAR . 3) (VAR . 4))
(RSEND (CONST METHOD . LENGTH1) (IVAR . 1) (VAR . 3))
(JUMP 199)
```

Figure 8-9. Stmtgraph with Dead Definitions Removed.
The dead eq statement was finally removed.

The Move Eliminator again finds moves it can remove; it removes two move statements from the stmtgraph. The move from ivar1 to var5 is not removed because the Move Eliminator only considers moves between local variables.

```
(ENTER)
(MOVE (VAR . 5) (IVAR . 1))
(PRIMITIVE (VAR . 4) + (CONST 1 . 1) (ARG . 0))
(CONDITION BNNIL (VAR . 5) 194)
(REPLY (VAR . 4))
(LABEL 199)
(EXIT)
(LABEL 194)
(RSEND (CONST METHOD . LENGTH1) (IVAR . 1) (VAR . 4))
(JUMP 199)
```

Figure 8-10. Stmtgraph with Moves Removed (Again).
Two of the move statements introduced by the Fork Merger are now gone.

The Touch Eliminator again does nothing, while the Dataflow Optimizer changes the bnnil to use ivar1 directly instead of var5, thus making the move statement dead.

```
(ENTER)
(MOVE (VAR . 5) (IVAR . 1))
(PRIMITIVE (VAR . 4) + (CONST 1 . 1) (ARG . 0))
(CONDITION BNNIL (IVAR . 1) 194)
(REPLY (VAR . 4))
(LABEL 199)
(EXIT)
(LABEL 194)
(RSEND (CONST METHOD . LENGTH1) (IVAR . 1) (VAR . 4))
(JUMP 199)
```

Figure 8-11. Stmtgraph after Second Dataflow Optimization.
Since the + primitive is known not to change values of instance variables, the Dataflow Optimizer can safely change bnnil's variable to ivar1, thus making var5 dead.

The optimizations remaining in the second pass are unable to make any improvements. Nevertheless, since the second pass did yield changes, a third pass through the optimizations is made. This time the Dead Definition Eliminator spots the dead move and eliminates it.

```
(ENTER)
(PRIMITIVE (VAR . 4) + (CONST 1 . 1) (ARG . 0))
(CONDITION BNNIL (IVAR . 1) 194)
(REPLY (VAR . 4))
(LABEL 199)
(EXIT)
(LABEL 194)
(RSEND (CONST METHOD . LENGTH1) (IVAR . 1) (VAR . 4))
(JUMP 199)
```

Figure 8-12. Optimized Stmtgraph.
The move of var5 is now gone, and no more optimizations can be made.

The Statement Optimizer makes one more pass through the optimizations without finding any changes, so it returns the stmtgraph shown in Figure 8-12. The Statement Postprocessor does not make any changes to the stmtgraph, so that stmtgraph is passed to the Instruction Generator.

The Instruction Generator calls the Variable Allocator to allocate local variables and create the varinfo record. Only var4 is referenced, and it is allocated to register R2, so the varinfo record is as shown in Figure 8-13.

```
(VARINFO
 (NVLOCS        NIL)
 (NARGS         1)
 (NIVARS        2)
 (IVARS-USED    T)
 (VARLOCS       (0) (1) (2) (3) (4 REG . 2) (5)))
```

Figure 8-13. Varinfo Record.
The only local variable is assigned to R2. Instance variables are used (ivars-used is true), but a context is not needed (nvlocs is nil).

Next the stmtgraph is linearized and compiled into instructions. The module that is the output of the Instruction Generator is shown in Figure 8-14. As with stmtgraphs, the module is too long and complicated to show in a figure; instead, the Instruction Printer was run on the module to show what it would have been if it had been output before any of the Assembly Code Generator's transformations are done.

```
MODULE PAIR__LENGTH1
        DC      MSG:LoadCode+20
        DC      {Class_PAIR},{Method_LENGTH1}
        INIT-VLOCS                          ;  0
        MOVE    [2,A3],R0                   ;  0.5
        XLATE   R0,A2,XLATE_OBJ             ;  1
        MOVE    1,R3                        ;  1.5
        ADD     R3,[3,A3],R2                ;  2
        MOVE    [3,A2],R1                   ;  2.5
        BNNIL   R1,^L001                    ;  3
        MOVE    [4,A3],R1                   ;  3.5
        BNIL    R1,^L002                    ;  4
        DC      MSG:ReplyConst+4            ;  5
        WTAG    R1,1,R3                     ;  6
        LSH     R3,-16,R3                   ;  6.5
        SEND2   R3,R0                       ;  7
        SEND    R1                          ;  7.5
        SEND    [5,A3]                      ;  8
        SENDE   R2                          ;  8.5
L001:   MOVE    [3,A2],R0                   ;  9
        CALL    Send_Node_Nr                ;  9.5
        DC      MSG:SendConst+7             ; 10
        SEND2   R1,R0                       ; 11
        DC      {Method_LENGTH1}            ; 12
        SEND    R0                          ; 13
        SEND    [3,A2]                      ; 13.5
        SEND    R2                          ; 14
        SEND    [4,A3]                      ; 14.5
        SENDE   [5,A3]                      ; 15
L002:   SUSPEND                             ; 16
        END
```

Figure 8-14. Initial Module.
This module lacks any unconditional branches (which are represented by edges of the module's digraph not shown in this Figure).

The Branch Inserter runs next and introduces an unconditional branch. Afterwards, the Variable Initializer runs and converts the `init-vlocs` pseudo-instruction into code to initialize context variables; there are none, so the `init-vlocs` pseudo-instruction is simply removed, resulting in the module shown in Figure 8-15.

```
MODULE PAIR__LENGTH1
        DC      MSG:LoadCode+20
        DC      {Class_PAIR},{Method_LENGTH1}
        MOVE    [2,A3],R0               ;  0
        XLATE   R0,A2,XLATE_OBJ         ;  0.5
        MOVE    1,R3                    ;  1
        ADD     R3,[3,A3],R2            ;  1.5
        MOVE    [3,A2],R1               ;  2
        BNNIL   R1,^L001                ;  2.5
        MOVE    [4,A3],R1               ;  3
        BNIL    R1,^L002                ;  3.5
        DC      MSG:ReplyConst+4        ;  4
        WTAG    R1,1,R3                 ;  5
        LSH     R3,-16,R3               ;  5.5
        SEND2   R3,R0                   ;  6
        SEND    R1                      ;  6.5
        SEND    [5,A3]                  ;  7
        SENDE   R2                      ;  7.5
        BR      ^L002                   ;  8
L001:   MOVE    [3,A2],R0               ;  9
        CALL    Send_Node_Nr            ;  9.5
        DC      MSG:SendConst+7         ; 10
        SEND2   R1,R0                   ; 11
        DC      {Method_LENGTH1}        ; 12
        SEND    R0                      ; 13
        SEND    [3,A2]                  ; 13.5
        SEND    R2                      ; 14
        SEND    [4,A3]                  ; 14.5
        SENDE   [5,A3]                  ; 15
L002:   SUSPEND                         ; 16
        END
```

Figure 8-15. Module before Instruction Optimization.
This module is already correct (it does not need any branches to be extended), but it can still be optimized.

Next the Instruction Optimizer and the Long Branch Handler are run. The module does not have any out-of-range short branches, so the Long Branch Handler does nothing. On the other hand, the Instruction Optimizer does merge two pairs of SEND instructions to yield, at last, the final output shown in Figure 8-16. Figure 8-17 shows what would have been output had all nonessential optimizations been turned off.

```
MODULE PAIR__LENGTH1
        DC      MSG:LoadCode+18
        DC      {Class_PAIR},{Method_LENGTH1}
        MOVE    [2,A3],R0               ;  0
        XLATE   R0,A2,XLATE_OBJ         ;  0.5
        MOVE    1,R3                    ;  1
        ADD     R3,[3,A3],R2            ;  1.5
        MOVE    [3,A2],R1               ;  2
        BNNIL   R1,^L001                ;  2.5
        MOVE    [4,A3],R1               ;  3
        BNIL    R1,^L002                ;  3.5
        DC      MSG:ReplyConst+4        ;  4
        WTAG    R1,1,R3                 ;  5
        LSH     R3,-16,R3               ;  5.5
        SEND2   R3,R0                   ;  6
        SEND    R1                      ;  6.5
        SEND2E  [5,A3],R2               ;  7
        BR      ^L002                   ;  7.5
L001:   MOVE    [3,A2],R0               ;  8
        CALL    Send_Node_Nr            ;  8.5
        DC      MSG:SendConst+7         ;  9
        SEND2   R1,R0                   ; 10
        DC      {Method_LENGTH1}        ; 11
        SEND    R0                      ; 12
        SEND2   [3,A2],R2               ; 12.5
        SEND    [4,A3]                  ; 13
        SENDE   [5,A3]                  ; 13.5
L002:   SUSPEND                         ; 14
        END
```

Figure 8-16. Final Output.

This is the MDP assembly code into which the Concurrent Smalltalk source code in Figure 8-1 compiles.

```
MODULE PAIR__LENGTH1
        DC      MSG:LoadCode+35
        DC      {Class_PAIR},{Method_LENGTH1}
        MOVE    4,R0                    ;  0
        CALL    New_Context             ;  0.5
        MOVE    NIL,R0                  ;  1
        MOVE    R0,[5,A1]               ;  1.5
        MOVE    R0,[6,A1]               ;  2
        MOVE    R0,[7,A1]               ;  2.5
        MOVE    R0,[8,A1]               ;  3
        MOVE    [2,A3],R0               ;  3.5
        XLATE   R0,A2,XLATE_OBJ         ;  4
        MOVE    [5,A1],R3               ;  4.5
        MOVE    [3,A2],R2               ;  5
        EQ      R2,NIL,R1               ;  5.5
        MOVE    R1,[5,A1]               ;  6
        MOVE    [5,A1],R3               ;  6.5
        BF      R3,^L001                ;  7
        MOVE    [6,A1],R2               ;  7.5
        MOVE    1,R3                    ;  8
        ADD     R3,[3,A3],R3            ;  8.5
        MOVE    R3,[6,A1]               ;  9
        BR      ^L002                   ;  9.5
L001:   MOVE    [8,A1],R2               ; 10
        MOVE    1,R3                    ; 10.5
        ADD     R3,[3,A3],R3            ; 11
        MOVE    R3,[8,A1]               ; 11.5
        MOVE    [6,A1],R2               ; 12
        MOVE    [8,A1],R3               ; 12.5
        MOVE    [3,A2],R0               ; 13
```

```
        MOVE    [3,A2],R1               ; 13.5
        CALL    Send_Node_Nr            ; 14
        DC      MSG:SendConst+7         ; 15
        SEND2   R1,R0                   ; 16
        DC      {Method_LENGTH1}        ; 17
        SEND    R0                      ; 18
        SEND    [3,A2]                  ; 18.5
        SEND    [8,A1]                  ; 19
        SEND    [1,A1]                  ; 19.5
        SENDE   6                       ; 20
        WTAG    R1,6,R1                 ; 20.5
        MOVE    R1,[6,A1]               ; 21
L002:   MOVE    [7,A1],R3               ; 22
        MOVE    [6,A1],R0               ; 22.5
        MOVE    R0,[7,A1]               ; 23
        MOVE    [7,A1],R2               ; 23.5
        MOVE    [4,A3],R3               ; 24
        BNIL    R3,^L003                ; 24.5
        DC      MSG:ReplyConst+4        ; 25
        WTAG    R3,1,R1                 ; 26
        LSH     R1,-16,R1               ; 26.5
        SEND2   R1,R0                   ; 27
        SEND    R3                      ; 27.5
        SEND    [5,A3]                  ; 28
        SENDE   [7,A1]                  ; 28.5
L003:   MOVE    [5,A1],R1               ; 29
        MOVE    [6,A1],R3               ; 29.5
        MOVE    [7,A1],R3               ; 30
        MOVE    [8,A1],R3               ; 30.5
        CALL    Free_Context            ; 31
        SUSPEND                         ; 31.5
        END
```

Figure 8-17. Unoptimized Output.
This is the output of the compiler from the source code in Figure 8-1 when all nonessential optimizations are turned off.

Other Examples

Figure 8-18 is an example of an accessor method that is automatically defined when a class (in this instance the class pair from Figure 8-1) is defined.

Figure 8-19 contains a listing of a larger Concurrent Smalltalk method together with some supporting methods. InsertKey compiles into 142 words and uses 3 context variables when optimization is turned on. When optimization is off, it cannot be compiled at all because it uses too many context variables; but if it could be compiled, it would require 193 words and 18 context variables.

```
MODULE PAIR__CAR
        DC      MSG:LoadCode+10
        DC      {Class_PAIR},{Method_CAR}
        MOVE    [2,A3],R0               ;  0
        XLATE   R0,A2,XLATE_OBJ         ;  0.5
        MOVE    [3,A3],R3               ;  1
        BNIL    R3,^L001                ;  1.5
        DC      MSG:ReplyConst+4        ;  2
        WTAG    R3,1,R2                 ;  3
        LSH     R2,-16,R2               ;  3.5
        SEND2   R2,R0                   ;  4
        SEND    R3                      ;  4.5
        SEND    [4,A3]                  ;  5
        SENDE   [2,A2]                  ;  5.5
L001:   SUSPEND                         ;  6
        END
```

Figure 8-18. An Accessor Method.

This method returns the value of one of the instance variables of its instance object.

```
(method bnode p.i.e. (p i e) ()
     (set parent p)
     (set indices i)
     (set entries e)
     (return self))
(method bnode p. (p) ()
     (set parent p))

(Method Bnode insertKey (nkey) (node-and-key new-root)
     (if (leaf? self)
      (begin (set indices (insert indices (find-place indices nkey 0)
                          nkey))
             (if (>= (length indices) 10)
              (begin (set node-and-key (split-node self))
                   (if (eq parent '())
                    (begin (set new-root (new bnode))
                         (p.i.e. new-root '()
                              (cons (rest node-and-key) '())
                              (cons self (cons (first node-and-key) '())))
                         (set parent new-root)
                         (return new-root))
                    (insertNode parent (first node-and-key)
                              (second node-and-key))))
              (return indices)))
        (begin (insertKey (select-child self nkey) nkey)
             (return self))))

(method bnode leaf? () ()
  (eq entries '()))
```

Figure 8-19. A Nontrivial Concurrent Smalltalk Method.

The above method, courtesy of Andrew Chien, is a part of a balanced tree handler.

Chapter 9. Conclusion

Results

The Optimist compiler does work as expected and in a reasonable amount of time. The code it produces is compact, and there are no obvious simple ways of significantly improving the code density or speed of the compiled methods.

I did successfully load the compiled methods into MDPSim, the MDP simulator [10], but I was unable to run them because the operating system services required by the methods were not yet ready. I will try to run the compiled methods when Brian finishes his JOSS operating system [12].

Optimizations

Although many of the optimizations used by the Optimist compiler are generally known, they have usually been applied to compilers for conventional processors. The issues involved in compiling for the MDP are quite different from compiling for conventional processors. For instance, keeping the code size small and dealing with unavailable values (futures) are critical issues on the MDP, while they are either not particularly important or not relevant on most conventional processors. On the other hand, loops and pointers are important areas (and stumbling blocks) of optimizations for conventional processors, while they are not a major concern on the MDP (the current version of Concurrent Smalltalk does not even have loops!).

The new ideas in the Optimist compiler include juxtaposing existing optimizations to fit the MDP. Also, I did include optimizations in the compiler that I had not seen before. These include the Fork and Join Mergers which can make unusual improvements to the generated code, the `touch` optimizations, the Move Eliminator which eliminates `move` statements not caught by standard copy propagation techniques such as the one given on page 636 of [2] (Figure 5-3), the Linearizer, and a myriad of code generator optimizations, many of them involving MDP idiosyncrasies such as word-aligning `DC`s, avoiding faults during message sends, preventing limbo variables, and accomodating long branches that are quite annoying because they trash an important register when used.

Effects of Optimizations

Having the various optimizations in the compiler is certainly worthwhile. After examining the compiler's output it becomes apparent that the optimizations are more than a luxury—they are essential to the successful use of Concurrent Smalltalk on the MDP. The compiler's optimizations reduce the amount of code output by anywhere between 20% and 60% (or even more in some cases) compared to the Optimist compiler's output with all nonessential optimizations disabled. The larger Concurrent Smalltalk methods such as the one in Figure 8-19 cannot be compiled at all without the optimizations because there are not enough context variables on the MDP available for use as the method's temporaries.

All of the compiler's optimizations are important to some extent, and there is a symbiosis effect among the various optimizations; for example, the Move Eliminator and the Fork and Join Mergers reinforce each other, and removing one would drastically reduce the effectiveness of the other. However, there are a few that are especially useful:

Space Optimizations

• The preprocessor and the Join Merger are almost always successful in reducing the number of `reply` statements in the compiled code to either zero or one. Since the code for a `reply` statement does require a significant amount of space, these optimizations are very helpful.

• The Register Allocator is very good at allocating variables to registers, often being able to allocate from 80% to 100% of the local variables to registers. Such unusually good performance on with so little registers is due to the Lisp-like nature of Concurrent Smalltalk—almost all temporaries are live for only one statement. Allocating variables in registers results in significant space savings.

• The Frame Handler produces consistently large space savings in the generated code. For the reasons presented in its description, the Stmt Compiler often requests a guarantee that certain variables' values are available; if it is unable to obtain such a guarantee, it touches these variables. When the Frame Handler optimizations are turned off, a significant amount of code is spent touching arguments before `csends`, `rsends`, `replys`, and even `primitives`.

• The Move Eliminator complements the other space optimizations by removing the extra `move` instructions introduced by the Front End and various other optimizations.

• The Fork and Join Mergers, when they are successful, are capable of eliminating large chunks of code. The programming style of Concurrent Smalltalk encourages writing expressions that can be merged by the Fork Merger, such as

```
(if test (expr1 (calculate a b c) ...)
  (expr2 (calculate a b c) ...))
```

where the two calls to calculate on both sides of the if can be merged, and by the Join Merger, such as

```
(if test (calculate (expra ...) (exprb ...) (exprc ...))
  (calculate (exprd ...) (expre ...) (exprf ...)))
```

where the two calls to calculate on both sides of the join can also be merged.

Speed and Data Space Optimizations

• Tail Forwarding is the most important speed and data space optimization. It reduces the data space required by many tail-recursive programs from linear to constant, which is extremely important in the limited memory environment of the MDP. Tail Forwarding also reduces the space required by the method code because an `rsend` (a tail forwarded method call) takes much less room than a normal `csend` followed by a `reply` statement. In many small methods such as the one in Figure 8-1 the speed savings are compounded because the conversion of a `csend` into an `rsend` can eliminate the need to allocate a context object for the method, saving a lot of time on entry to the method.

• Another very important speed optimization is the elimination of the calls to allocate and deallocate a context object if the method does not have any context variables, which happens when all local variables can fit in registers and there are no `csends` with local variables as targets in the optimized stmtgraph. This condition holds for all of the accessor methods for a class as well as many other simple methods (see Chapter 8). The speed gains can be considerable because allocating and deallocating a context takes a few dozen instruction cycles in the operating system, which for small methods is significantly more than the time spent executing the actual method code.

- Register allocation and compaction of context variables by the Variable Allocator is an important data space saving optimization. It is usually quite successful—for the method in Figure 8-19, the number of variables in the context was reduced from 18 (which was more than the MDP can address) to just 3.

Programmer Convenience Optimizations

- The Dataflow Optimizer and Constant Folder are very good at eliminating dead code and folding various constant expressions. These optimizations are important when symbolic constant expressions or debugging code controlled by a constant flag is included in the source code. A Concurrent Smalltalk programmer can rest easier and write cleaner programs when he knows that introducing debugging statements controlled by a constant will not affect the performance of his code when debugging is turned off.

Impact on MDP Project

The Optimist compiler fills one of the few remaining missing links in the abstraction layers of the MDP project. The layers above the compiler include research on programing in Concurrent Smalltalk, while the layers below the compiler are the JOSS operating system; the MDP Architecture; MDPSim, the instruction simulator; and various lower-level hardware simulators.

As expected, implementing the compiler made apparent minor deficiencies in the layers around the compiler. Most of these deficiencies have been corrected by the time of this writing, and this thesis lists only the new information. On the higher level, the syntax of Concurrent Smalltalk was changed to include the `reply` and `return` statements, implicit tail forwarding (instead of the original explicit `forward requester` construct which led to numerous programmer errors), and a more orthogonal syntax for invoking variable methods and passing them as parameters to other methods (the new syntax is based on the syntax of the Scheme language [1], which treats functions as first-class data objects).

Memory Space

The deficiencies on the lower level, the MDP Architecture and hardware, were more serious. Implementing the compiler required the addition of the `CFUT` data type to be able to mark variables as unavailable (the implementation of `CFUT`s in the MDP still isn't done quite right, as a few holes remain in the current type-checking system). However, the most serious problem raised by the compiler is the lack of memory space on the MDP. The MDP only contains 4096 words of RAM, and much of that space is taken by the operating system. Seemingly small methods such as the one listed in Figure 8-19 can easily compile into a significant fraction of the MDP's memory space. Furthermore, the MDP's queues are currently set to accept messages up to 128 words in length; under the current operating system, the method in Figure 8-19 could not even be sent to a MDP!

A solution to the memory space is needed before the MDP project can continue. Two alternatives appear to be reasonable solutions. One is to increase the MDP's memory to a reasonable amount such as 16384 words. That amount might be sufficient to fit useful programs; to confirm or deny this claim an actual useful program written in Concurrent Smalltalk is needed. The second alternative is to use slow off-chip memory to supplement the MDP's on-chip memory. This alternative would suffer from the problem of deciding what data should be placed on-chip and what off-chip, so if an external memory interface is added to the MDP, it might actually be better to convert all of the MDP's on-chip memory into one large cache for the off-chip memory. If this route is chosen, the MDP's memory space would

be limited only by its 1-megaword addressing range, but the MDP would lose some performance.

Grain Size

The initial goal for the MDP project was to have the MDP execute finely grained parallel programs containing about twenty instructions per method. When only the output of the compiler is examined, this goal appears to have been met (see Figure 8-16). Unfortunately, it turns out that for various reasons the system calls New_Context, Free_Context, and especially the method dispatcher (the routine that runs whenever a MDP receives a SEND method-call message) and the fault handling routines are quite long, taking dozens of instruction times to execute. Thus, if the time spent executing operating system code is taken into account, the grain size is approximately 50 to 100 instructions (50 instructions ≈ 5 μs at 10^7 instructions/sec), which is still very good compared with other parallel computer efforts.

Future Improvements

The compiler is an evolving project, and I plan on improving it over time to fit the needs of the MDP project. The obvious additions to the compiler include support of global variables and lexically scoped blocks. These features were not included in this revision of the compiler because it is not clear how the operating system would support them.

Additional work can be done on the compiler's optimizations. One type of optimization that needs to be added is global optimizations such as inlining small methods when appropriate. It is good programming style to write little abstraction methods like leaf? in Figure 8-19 that perform a very simple task and return. Unfortunately, all such methods are currently compiled as method calls because the compiler only has access to one method at a time while compiling. Compiling these methods as calls often leads to a large waste of time as well as code space; for example, in the method insertKey, the call to leaf? could be replaced by an inline check of the entries variable, saving both space and time.

Another possible improvement is fine-tuning the register assignment process. Currently the register assigner works with conservative estimates of the maximum number of temporary registers required by a statement. These estimates could be improved by considering more data.

Finally, in some cases it might be desirable to keep a variable in memory for a part of its lifetime and in a register for another part. One simple approach to achieve this goal in many cases is to split a variable into two or more variables as in the example below:

```
(cset a (expression1 ...))
(expression2 a ...)
(cset a (expression3 ...))
(expression4 a ...)
```

can be changed to

```
(cset b (expression1 ...))
(expression2 b ...)
(cset c (expression3 ...))
(expression4 c ...)
```

Now b and c are live in disjoint intervals, so in the worst case they will be placed in the same context variable or the same register. It is possible, though, that one of them will be placed in memory and the other in a register, whereas in the original code a would then be placed in memory.

Summary

The Optimist compiler compiles Concurrent Smalltalk to the MDP assembly language. Although the compiler includes various optimizations such as dead code elimination, dataflow analysis, constant folding, move elimination, duplicate code merging, tail forwarding, use of register variables, and not allocating a context unless necessary, the size of the compiled code appears to be larger than was anticipated, and it is unlikely that application programs will fit (together with their data and the RAM-based portion of the operating system!) in the 4096 words available on the MDP prototype. Once the memory size problem is resolved, however, the future of the MDP project looks very bright, as it appears that the methods compiled by this compiler will have a reasonably small (20 to 100 instructions) grain size.

Appendix A. MDP Architecture Summary

This Appendix is a summary of the version 10 MDP Architecture document [9]. Many details have been simplified in order to keep this Appendix to a reasonable length.

Introduction

The Message-Driven Processor is a processing node for the J-Machine, a message-passing concurrent computer. The MDP is designed to provide support for fine-grained concurrent computation. Towards this goal the processor includes hardware for message queueing, low-latency message dispatching, and message sending. The same chip also contains a network interface and a router to allow the routing of messages throughout the network without any processor intervention.

The size of the MDP's register set is limited to minimize context-switching time. The memory is on the chip to improve performance and reduce the chip's pin count and the chip count for the concurrent computer. Having memory on chip allows more flexibility in the use of memory than in designs with off-chip memory. For example, a portion of memory may be designated as a two-way set-associative cache to be used by the XLATE instruction.

The MDP is also designed to efficiently support object-oriented programming. Every MDP word consists of 32 bits and a 4 bit tag that classifies the word as an integer, boolean, address, instruction, pointer, or other data. The MDP's four address registers include both base addresses and lengths, so all memory accesses are bounds checked. Normally the address registers point to objects, so, since absolute memory addressing is not allowed except by the operating system, memory references can only be made to objects relative to their beginnings. Having tags and no absolute references permits the use of garbage collection and transparent migration of objects to other MDP nodes on the network.

The MDP is almost completely message-driven. It is controlled by the messages arriving from the network that are automatically queued and processed. There are two priority levels to allow urgent messages to interrupt normal processing. There is also limited support for a background mode of execution when no messages are waiting in the queues.

Processor State

The processor state of the MDP is kept in a set of registers shown in Figure A-1. There are two independent copies of most registers registers for each of the two priorities of the MDP, allowing easy priority switches while keeping the integrity of the registers. The registers are symbolically represented as follows:

- `R0-R3` general-purpose data registers
- `A0-A3` address registers
- `ID0-ID3` ID registers
- `SR` status register
- `IP` instruction pointer register
- `TRP` trapped instruction register
- `SP` stack pointer register
- `QBM` queue base/limit register
- `QHL` queue head/tail register
- `TBM` translation base/mask register
- `NNR` node number register

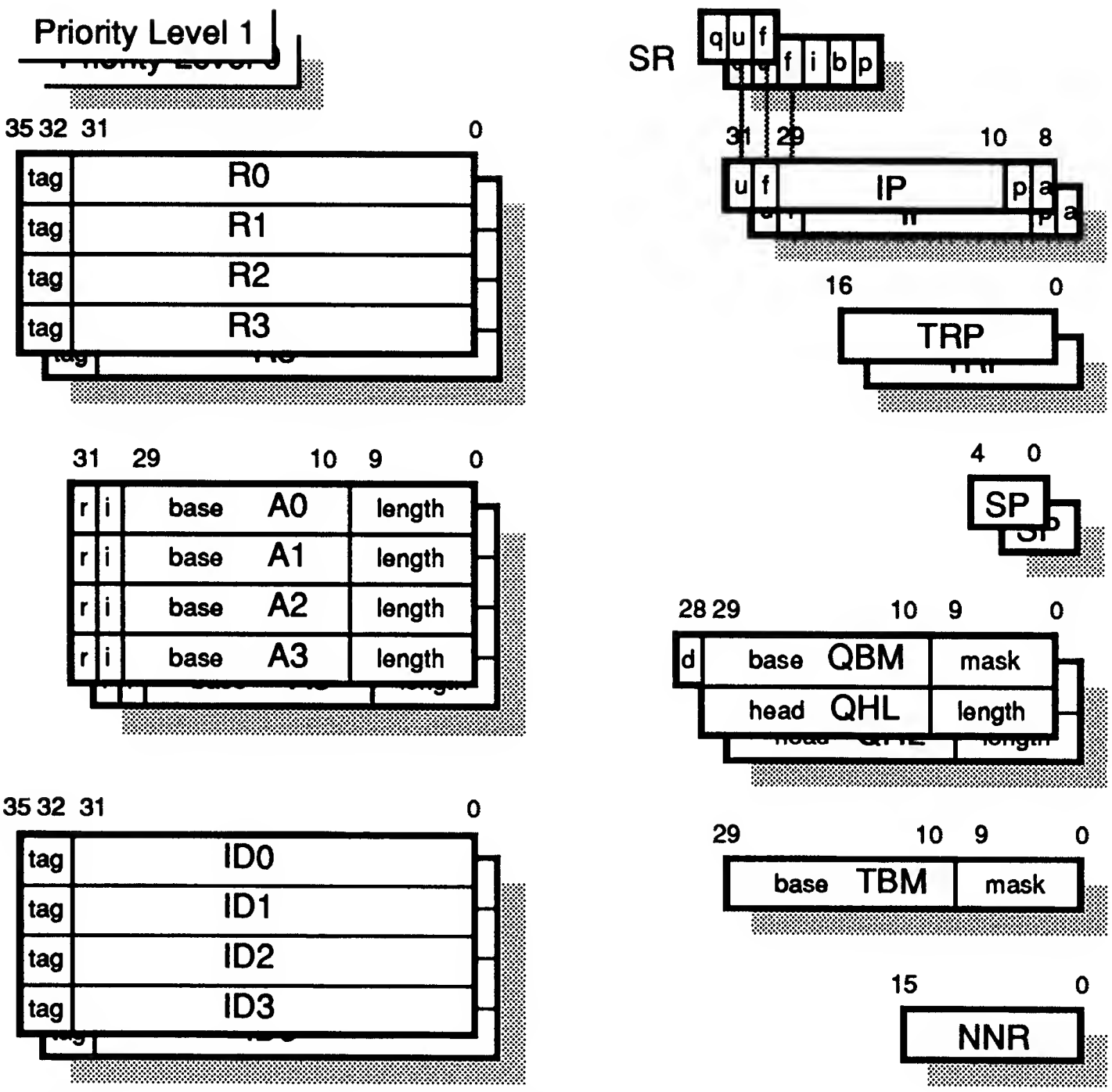


Figure A-1. The MDP Register Set.

Data Types

The following data types that may be used in a word are shown in Figure A-2. All data types except possibly FUT and CFUT may be moved, compared with EQ and NEQ, XLATEd and ENTERed, RTAGged, WTAGged, CHECKed, and executed. Executing a non-INST word causes it to be loaded into R0. Some data types allow additional operations, which are listed in detail in the description of the instruction set.

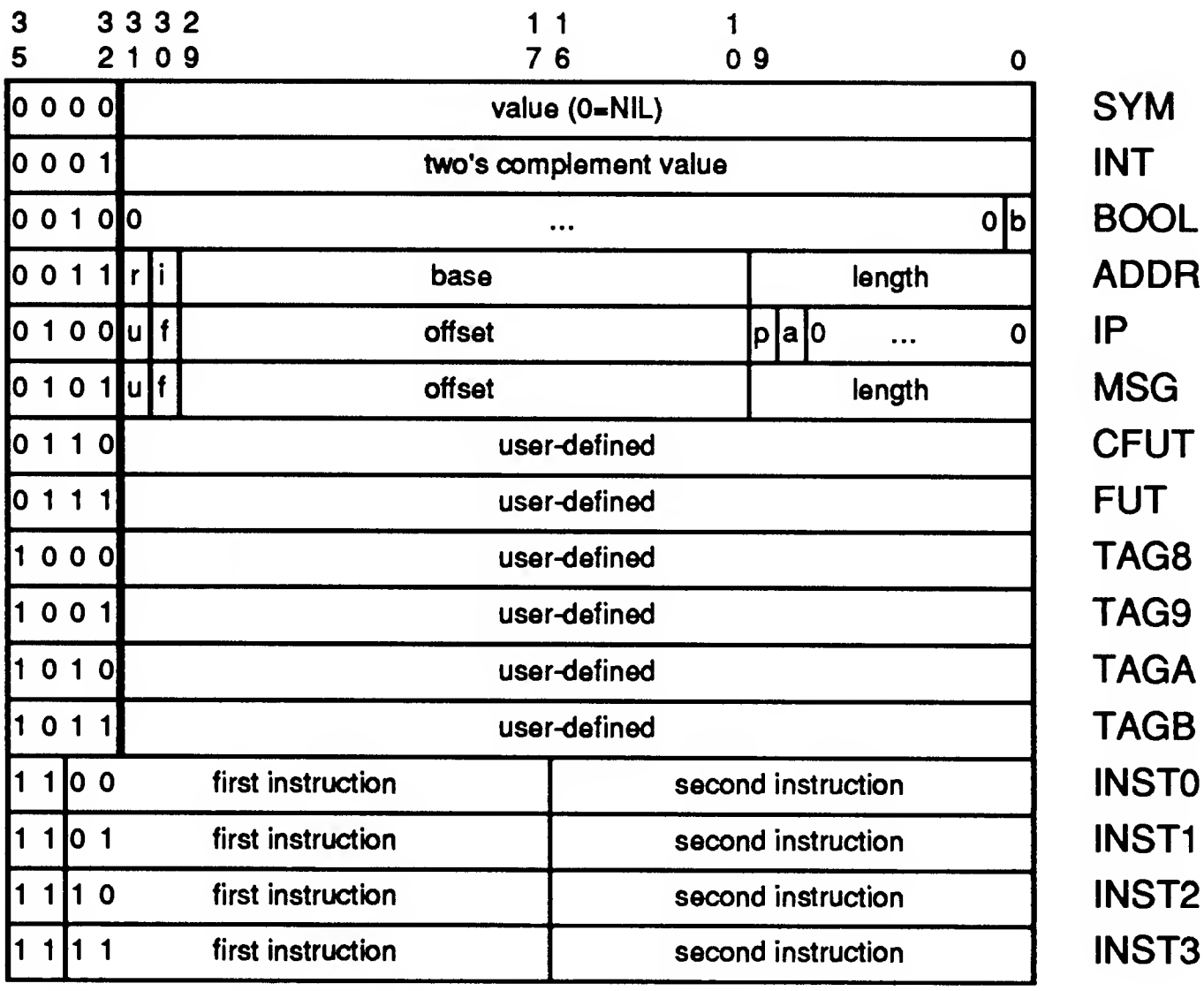


Figure A-2. The MDP Data Types.

- SYM contains an atomic symbol. EQUAL and NEQUAL are allowed on SYMbols. If the data portion of a symbol contains all zeroes, the word takes on the value of NIL.
- INT contains a two's complement integer between -2^{31} and $2^{31}-1$, inclusive. All arithmetic, logical, and comparison operations are allowed on INTs.
- BOOL contains a boolean value, which is either true (b=1) or false (b=0). MAX, MIN and all logical, and comparison operations are allowed on BOOLs. For purposes of MAX, MIN, and the comparisons, false is considered as less than true.
- ADDR contains a base/length pair that may be loaded into either one of the address registers or QBM, QHL, or TBM. The uses of bits 30 and 31 vary among these registers.
- IP contains a value appropriate for loading into the IP.
- MSG is the header of a message. It is similar to an IP.
- CFUT contains a context future. Almost all operations fault on context futures. They are not meant to be MOVEable. CFUTs are used as placeholders for unavailable values to be computed in parallel by other processes; an attempt to read a CFUT before its value is available will fault, and the operating system will suspend the current process until the value is available.
- FUT is a standard future. FUTs may be moved, and their tags may be read and written, but they may not participate in any primitive operations such as addition or checking for equality. As with CFUTs, an attempt to use a FUT in a primitive operation will cause a fault, and the operating system will have to provide the appropriate value for the FUT.
- TAG8 through TAGB are tags for software-defined words. They cause faults on all primitive operations except EQ, NEQ, BNIL, and BNNIL.
- INST0 through INST3 are tags for instructions. The two instructions in a word occupy a total of 34 bits, so two tag bits are also used to encode them.

Network Interface

Incoming messages are queued in *message queues* before being dispatched and processed. There are two message queues, one for each priority level. When a message arrives, register A3 is set up to point to it in the message queue, and execution begins at the address specified by the message header. A message may be processed as soon as its first word arrives; the processor does not wait until the entire message is present before processing it. Memory accesses to the message are checked to make sure that the processor does not try to access a word in the message before it arrives; if the processor tries to access a word too early, it waits until the word has arrived.

The SUSPEND instruction informs the hardware that the processing of the current message is done and that it should fetch the next message.

Message Transmission

The SEND, SEND2, SENDE, and SEND2E instructions are used to send messages. The first word sent specifies the node number of the destination node (i.e. the destination node's NNR value) in the low 16 bits. The SEND instruction will use the current node's NNR and the destination node number to find the relative offsets in the X and Y dimensions that the network controllers will use in routing the messages through the network. Bit 31 determines the priority at which the message will be sent over the network: 0 means priority level 0 and 1 means level 1. The priority of the message is independent of the priority of the process that is sending it.

The initial routing word is followed by a number of words which the network delivers verbatim to the destination node. The network does not examine the contents of these words. The message is terminated by a SENDE or SEND2E instruction, which send the last one or two, respectively, words of it and inform the network to actually transmit the message. The first word that arrives at the destination node (the second word actually sent since the routing word is only used by the network and doesn't arrive at the destination node) must be tagged MSG. It contains the length of that message including that word but not including the routing word preceding it. It also contains the initial value of the IP at which execution is supposed to start. The destination node will fault MSG if this word is incorrect.

The total time between the first SEND and the SENDE should be as short as possible to avoid blocking the network. For the same reason, faults should be avoided while sending. There is a small (8 words or so) send buffer present. If a message exceeds the size of the send buffer, interrupts are internally disabled until the next SENDE.

Fault Processing

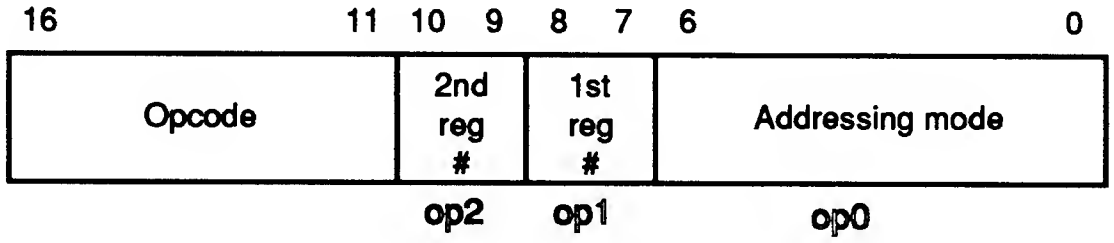
When a fault occurs, the current IP is incremented to the next instruction and pushed on the current priority's stack. The instruction that caused the fault is saved in the TRP register and the IP is then fetched from the memory location whose address is equal to the fault number.

Instruction Encoding

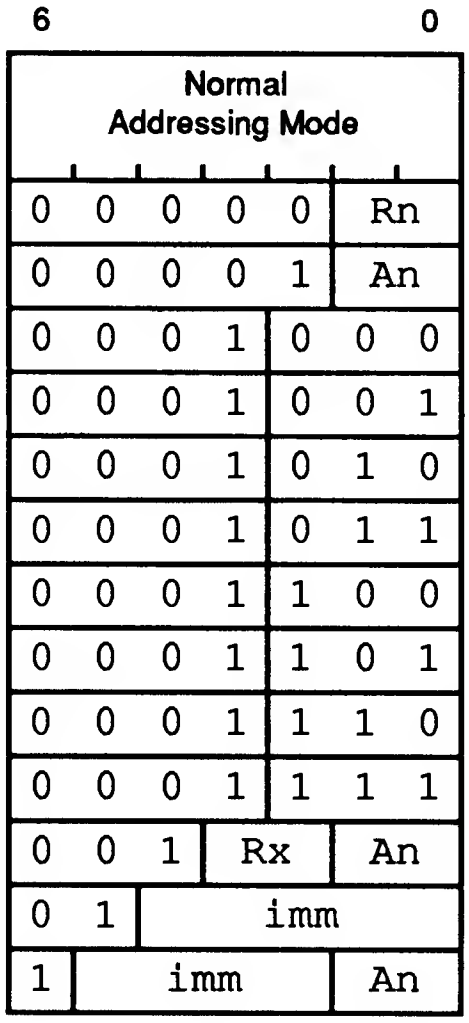
The program executed by the MDP consists of instructions and constants. A constant is any word not tagged INST0 through INST3 that is encountered in the instruction stream. When a constant word is encountered, that word is loaded into R0 and execution proceeds with the next word (the assembler syntax for including a word in the code stream is DC).

Every instruction is 17 bits long. Two 17-bit instructions are packed into a word. Since a word has only 32 data bits, two tag bits are also used to specify the instructions. The instruction in the high part of the word is executed first, followed by the instruction in the low part of the word. As a matter of convention, if only one instruction is present in a word, it should be placed in the high part, and the low part of the word set to all zeros.

The format of an instruction is as follows:



The *opcode* field specifies one of 64 possible instructions. The other fields specify three operands; instructions that don't require three operands ignore some of the operand fields. Operands 1 and 2 must be data registers; their numbers (0 through 3) are encoded in the *1st reg #* and *2nd reg #* fields. Operand 2, if used, is always the destination of an operation and operand 1, if used, is always a source.



Syntax	Addressing Mode
Rn	Data register Rn
An	Address register An
NIL	Immediate constant NIL (SYM:0)
FALSE	Immediate constant FALSE (BOOL:0)
TRUE	Immediate constant TRUE (BOOL:1)
$80000000	Immediate constant INT:$80000000
$FF	Immediate constant INT:$000000FF
$3FF	Immediate constant INT:$000003FF
$FFFF	Immediate constant INT:$0000FFFF
$FFFFF	Immediate constant INT:$000FFFFF
[Rx,An]	Offset Rx in object An
imm	Immediate imm (signed)
[imm,An]	Offset imm (unsigned) in object An

Figure A-3. The MDP Normal Addressing Modes.
The immediate constants are eight immediate values outside the range INT:-16..INT:15. They are provided for convenience and code density improvement. The $FF and $FFFF constants are useful for masking bytes and words, while the $3FF and $FFFFF constants may be used for masking lengths and addresses.

Operand 0 can be used as a source or a destination in an instruction. It can hold two possible encodings. A normal instruction has op0 address mode encodings as shown in Figure A-3. The register-oriented op0 mode is used instead by a few instructions such as PUSH and POP;

the MOVE instructions allow both address mode encodings. If an instruction uses the register-oriented op0, the encodings are as in Figure A-4.

6	Register-Oriented Addressing Mode					0	Syntax		Addressing Mode
P	0	0	0	0	Rn		Rn	Rn`	Data register Rn
P	0	0	0	1	An		An	An`	Address register An
P	0	0	1	0	IDn		IDn	IDn`	ID register IDn
-	0	0	1	1	-	-			Unused (ILGADRMD fault)
P	0	1	0	0	0	0	QBM	QBM`	Queue Base/Mask register
P	0	1	0	0	0	1	QHL	QHL`	Queue Head/Length register
P	0	1	0	0	1	0	IP	IP`	Instruction Pointer register
P	0	1	0	0	1	1	SP	SP`	Stack Pointer register
P	0	1	0	1	0	0	TRP	TRP`	Trapped Instruction register
-	0	1	0	1	0	1	TBM		Translation Base/Mask register
-	0	1	0	1	1	0	NNR		Node Number register
-	0	1	0	1	1	1			Unused (ILGADRMD fault)
-	0	1	1	0	0	0	P		Priority Level flag
-	0	1	1	0	0	1	B		Background Execution flag
-	0	1	1	0	1	0	I		Interrupt flag
P	0	1	1	0	1	1	F	F`	Fault flag
P	0	1	1	1	0	0	U	U`	Unchecked flag
P	0	1	1	1	0	1	Q	Q`	A3 Queue flag
-	0	1	1	1	1	0			Unused flag (ILGADRMD fault)
-	0	1	1	1	1	1			Unused flag (ILGADRMD fault)
-	1	0	0	-	-	-			Unused (ILGADRMD fault)
-	1	0	1	-	-	-			Unused (ILGADRMD fault)
-	1	1	0	-	-	-			Unused (ILGADRMD fault)
-	1	1	1	IOREGn			IOREGn		I/O register IOREGn

Figure A-4. The MDP Register Oriented Addressing Modes.
P represents the priority of the register being accessed, and is relative to the current priority. 0 indicates the current priority, while 1 indicates the other priority. The assembler syntax for specifying a register belonging to the other priority is the register name followed by a backquote (`). The I/O register mode is reserved to provide access to various other registers that will be needed to interface the MDP to I/O devices.

Instruction Set Summary

The instructions supported by the MDP are summarized in Table A-1. The Types column specifies the types on which the instruction operates; if the arguments have different types, the instruction faults. Almost all instructions with the notable exception of MOVE to memory fault when any of their operands are tagged CFUT.

Table A-1. MDP Instructions

Instruction		Brief Description	Types
General Movement and Type Instructions			
MOVE	Src,Rd	Rd ← Src	All
MOVE	Rs,Dst	Dst ← Rs	All
RTAG	Src,Rd	Rd ← INT:tag(Src)	All
WTAG	Rs,Src,Rd	Rd ← Src:Rs	All
PUSH	Src	Push Src on the stack	All
POP	Dst	Pop the stack into Dst	All
CHECK	Rs,Src,Rd	Rd ← BOOL:tag(Rs)=Src	All
Arithmetic and Logical Instructions			
NEG	Src,Rd	Rd ← -Src	INT
ADD	Rs,Src,Rd	Rd ← Rs+Src	INT
SUB	Rs,Src,Rd	Rd ← Rs-Src	INT
CARRY	Rs,Src,Rd	Rd ← Carry from the addition of Rs and Src	INT
MAX	Rs,Src,Rd	If Rs≥Src then Rd ← Rs else Rd ← Src	INT, BOOL
MIN	Rs,Src,Rd	If Rs≤Src then Rd ← Rs else Rd ← Src	INT, BOOL
MUL	Rs,Src,Rd	Rd ← Rs*Src	INT
MULH	Rs,Src,Rd	Rd ← High 32 bits of 64-bit unsigned product of Rs and Src	INT
ASH	Rs,Src,Rd	Rd ← Rs shifted left arithmetically by Src bits	INT
LSH	Rs,Src,Rd	Rd ← Rs shifted left logically by Src bits	INT
ROT	Rs,Src,Rd	Rd ← Rs rotated left by Src (mod 32) bits	INT
NOT	Src,Rd	Rd ← NOT Src	INT, BOOL
AND	Rs,Src,Rd	Rd ← Rs AND Src	INT, BOOL
OR	Rs,Src,Rd	Rd ← Rs OR Src	INT, BOOL
XOR	Rs,Src,Rd	Rd ← Rs XOR Src	INT, BOOL
LT	Rs,Src,Rd	Rd ← BOOL:Rs<Src	INT, BOOL
LE	Rs,Src,Rd	Rd ← BOOL:Rs≤Src	INT, BOOL
GT	Rs,Src,Rd	Rd ← BOOL:Rs>Src	INT, BOOL
GE	Rs,Src,Rd	Rd ← BOOL:Rs≥Src	INT, BOOL
EQUAL	Rs,Src,Rd	Rd ← BOOL:Rs=Src	SYM, INT, BOOL
NEQUAL	Rs,Src,Rd	Rd ← BOOL:Rs≠Src	SYM, INT, BOOL
EQ	Rs,Src,Rd	Rd ← BOOL:Rs=Src (Pointer comparison only)	All
NEQ	Rs,Src,Rd	Rd ← BOOL:Rs≠Src (Pointer comparison only)	All
Network Instructions			
SEND	Src	Send Src onto the network	All
SENDE	Src	Send Src onto the network and terminate message	All
SEND2	Rs,Src	Send Rs and Src onto the network	All
SEND2E	Rs,Src	Send Rs and Src onto the network and terminate message	All

Associative Lookup Table Instructions

XLATE	Rs,Dst,C	Dst ← associative lookup in the associative lookup table of Rs	All
ENTER	Src,Rs	Enter (Src, Dst) into the associative lookup table	All
PROBE	Src,Rd	Rd ← BOOL:Src is in the associative lookup table	All
PURGE	Rs	Remove Rs from the associative lookup table	All

Special Instructions

INVAL		Invalidate all relocatable address registers	
SUSPEND		Terminate current process and fetch another message	
CALL	Src	Call system routine numbered Src	
RES	Src	IP ← Src	IP

Branches

BR	Src	Branch forward Src words	
BNIL	Rs,Src	Branch forward Src words if Rs is NIL	All
BNNIL	Rs,Src	Branch forward Src words if Rs is not NIL	All
BF	Rs,Src	Branch forward Src words if Rs is false	BOOL
BT	Rs,Src	Branch forward Src words if Rs is true	BOOL
BZ	Rs,Src	Branch forward Src words if Rs is zero	INT
BNZ	Rs,Src	Branch forward Src words if Rs is non-zero	INT

Appendix B. Operating System Interface

Registers

Methods generated by the compiler use the data, address, ID, and NNR registers. The registers are used for the purposes listed in Table B-1.

Table B-1. Compiler Register Usage

Register	Use
R0, R1, R2, R3	Temporaries used by the method code.
A0, ID0	Pointer to method that is executing. Set up by the operating system.
A1, ID1	Pointer to the context object or NIL if there is none.
A2, ID2	Pointer to the instance object.
A3, ID3	Pointer to the message that invoked the method.
NNR	Node number of this node.

When the incoming message arrives, the operating system should set A3, ID3, A0, and ID0 to their correct values and ID1 to NIL before calling the method. NNR should always be the node number of the node.

Faults

The operating system is expected to handle the common faults that arise during execution of user programs. In particular, the operating system should handle the following faults:

- TYPE and related faults: These faults occur if a primitive selector is invoked on a non-primitive receiver (i.e. + is attempted on a matrix). The operating system should send the appropriate message to the receiver object, wait for the reply, and store the result in the destination register of the instruction that caused the fault or emulate the behavior of the instruction if it was a condition.

- OVERFLOW: This fault occurs when an integer operation overflows the signed 32-bit range. In this case the operating system should call an extended precision integer package to determine the result and store the result in the destination register of the instruction that caused the fault.

- CFUT: This is probably the most common fault. It occurs whenever the method attempts to use the result of a SEND for which a REPLY has not yet been received. The operating system should suspend execution of the method until the reply does arrive and then restart the instruction that caused the fault.

- Any system faults not related to the executing method.

For all of the faults it is important that the operating system preserve all data and ID registers.

Data Structures

The data structures referenced by the compiler include the incoming message, the context, and the instance object, if any, of the method's class. The format of the incoming message is

shown in Figure B-3. Register A3 always points to the message that invoked the method during the execution of the method.

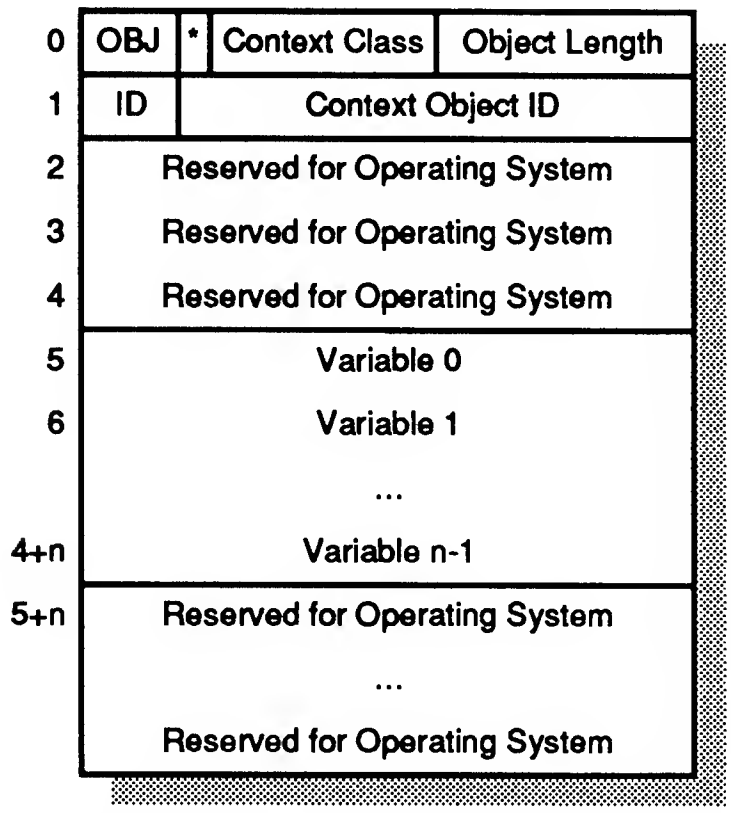


Figure B-1. Context Object Format

The context object is addressed by register A1. It contains the values of the method's local variables, the values of the registers when the method is suspended, as well as other miscellaneous information used by the operating system. Context Class is a constant that identifies this object as a context. * represents a few bits reserved for the operating system.

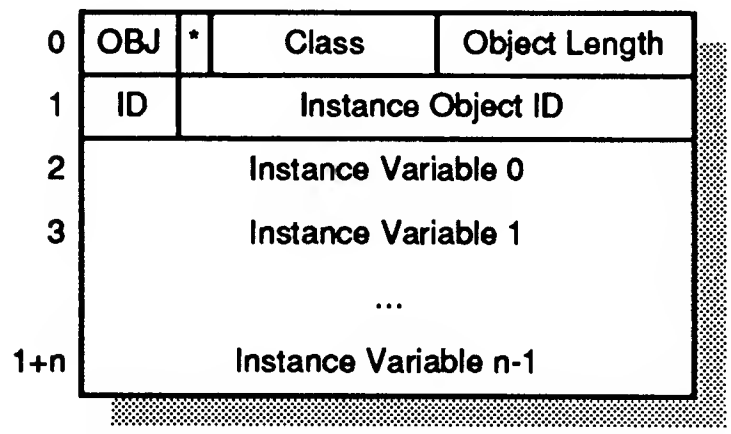


Figure B-2. Instance Object Format

The instance object, if any, is addressed by register A2. It contains the variables of the instance object on which the method is operating. Class is the instance object's class. * represents a few bits reserved for the operating system.

The context object has the format shown in Figure B-1. The only fields relevant to the compiler are the ID field that is sent in the reply ID field of all outgoing SEND messages that request a reply and the local variable fields that are used for local variable storage as well as slots into which values returned by REPLY messages are stored.

Unless the class is atomic, the method is called on an instance object. If the instance object is needed for execution of the method, the instance object ID is taken from receiver field of the caller's message and XLATEd into register A2. Instance variables are then addressed as offsets from register A2. The compiler permits the instance object to migrate away during the

execution of the method, in which case the next access of an instance variable will fault, and the operating system will have to bring the object back.

Message Formats

There are currently three kinds of messages used by the compiler: SEND messages, REPLY messages, and CODE messages. Their formats are shown in Figure B-3.

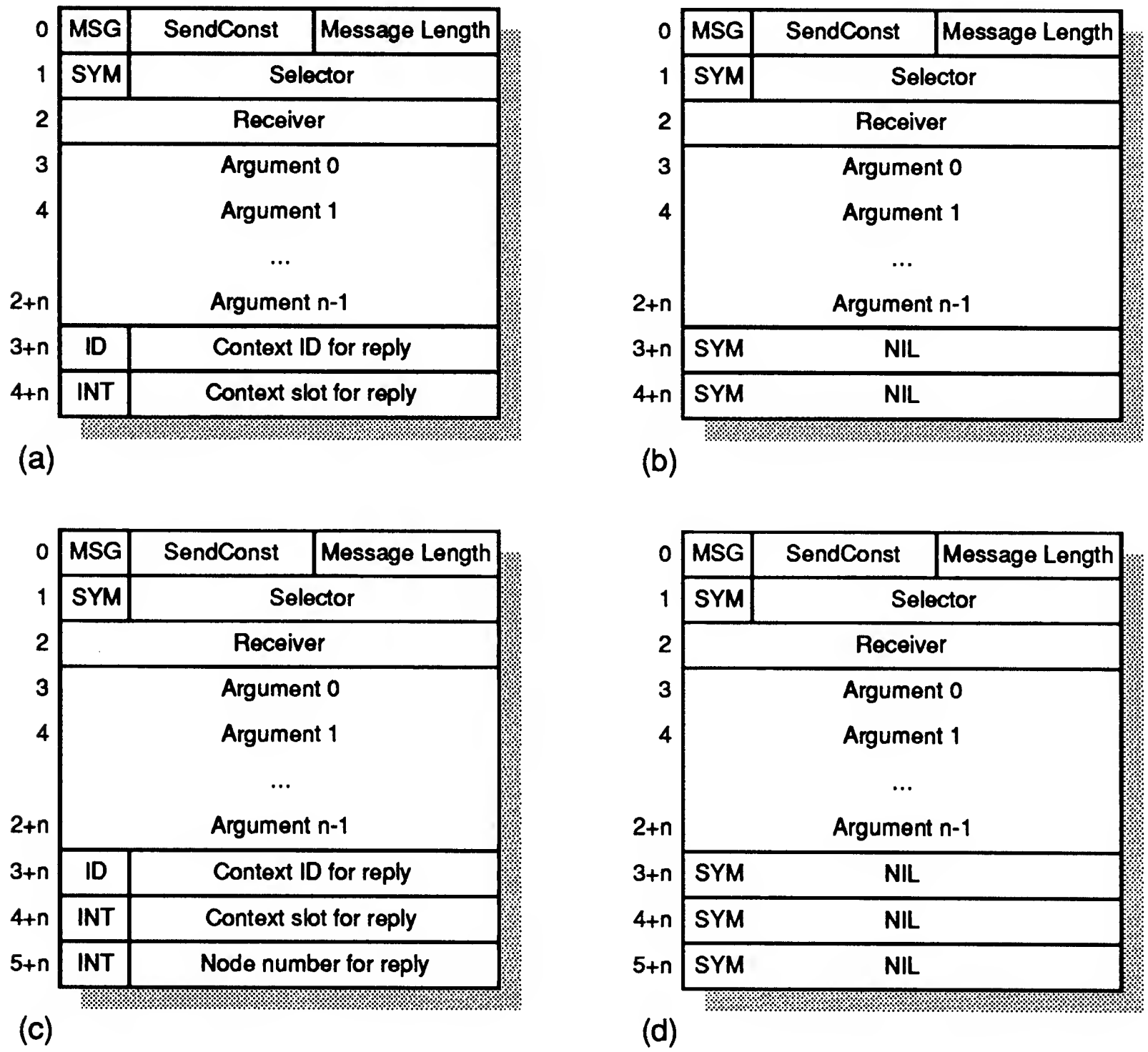


Figure B-3. SEND Message Formats

The format of a SEND message that expects a reply is shown in (a), while the SEND message in (b) does not expect a reply. The formats in (c) and (d) are the same except that an additional word is sent indicating the node number of the node to which the reply should be sent. Formats (c) and (d) are used when the *reply-node* flag is set.

A SEND message requests the execution of a method of the receiver object. The selector specifying the method is the second word of the message, and the receiver object is the third, followed by additional arguments, if any. If the caller expects a reply, it will indicate its ID and slot after the arguments. If the *reply-node* compiler flag is set, the caller includes an

additional word containing the caller's node number. The called method should then copy the ID and slot into the REPLY message along with the reply value. If no reply is expected, the last two (or three if *reply-node* is set) words of the message are set to NIL.

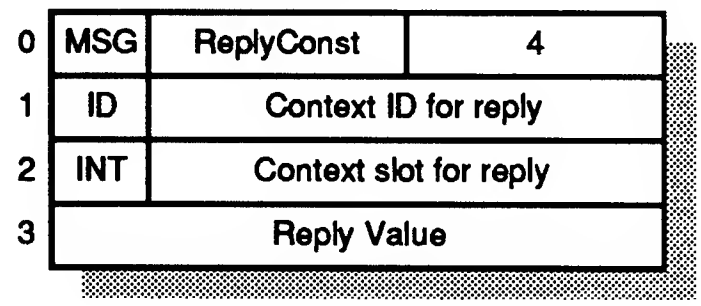


Figure B-4. REPLY Message Format
The REPLY message asks the operating system to store the reply value in the given slot (offset from the beginning of the context) of the context with the given ID. The caller should have reserved the slot for the reply by putting a CFUT in the specified slot.

A REPLY message is sent by REPLY and RETURN statements, as well as the implicit REPLY of the last expression of the method code. It communicates the result of the method back to the caller.

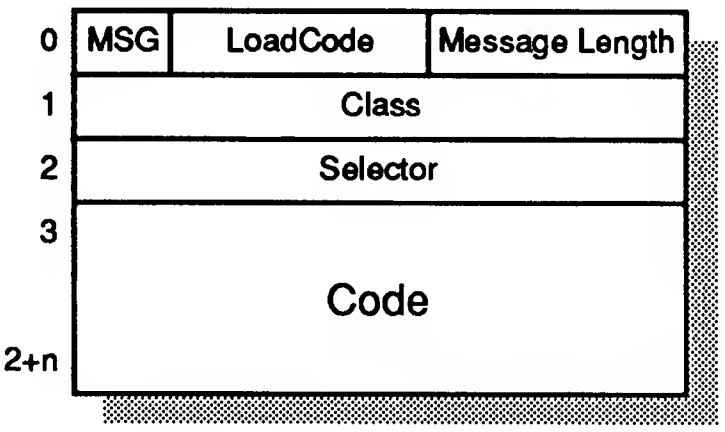


Figure B-5. CODE Message Format
The CODE message contains the executable code for a method along with the method's class and selector.

A CODE message is generated by the compiler. It specifies the code of a method. CODE messages are not directly manipulated by Concurrent Smalltalk methods other than methods that are part of the operating system.

System Calls

The system calls used by compiled code are listed in Table B-2.

Table B-2. System Call Specifications

New_Context

ENTRY:	R0 contains n, the number of local variables to allocate.
DESCRIPTION:	New_Context allocates a new context with n locals and returns the context's ID in ID1 and address in A1.
REGISTERS ALTERED:	R0,R1,A1,ID1.
TASK SWITCHING:	Yes.
EXIT:	ID1 contains the context object's ID. A1 points to the context object.

Free_Context

ENTRY:	ID1 contains the context object's ID. A1 points to the context object.
DESCRIPTION:	Free_Context deallocates the context and returns it to the free storage pool.
TASK SWITCHING:	Yes.
REGISTERS ALTERED:	R0,R1,A1,ID1.

New_Object

ENTRY:	R0 contains the number of instance variables of the class. R1 contains the class number.
DESCRIPTION:	New_Object allocates a new object of the specified class and returns it.
REGISTERS ALTERED:	R0,R1.
TASK SWITCHING:	Yes.
EXIT:	R0 contains the ID of the new object.

Send_Node_Nr

ENTRY:	R0 contains a receiver object.
DESCRIPTION:	Send_Node_Nr determines a node to which a SEND message involving the receiver object should be sent. If the receiver object is atomic, a random node is returned. If it is a true object, the operating system tries to guess the node on which the object currently resides.
REGISTERS ALTERED:	R0,R1.
TASK SWITCHING:	No.
EXIT:	R1 contains the node number to which to send the message.

Divide

ENTRY:	R0 contains the divisor. R1 contains the dividend.
DESCRIPTION:	Divide calculates R1//R0 and R1 mod R0 and stores the quotient in R0 and remainder in R1. An error occurs if R0=0. The quotient is rounded towards -∞, so 5//3 has quotient 1 and remainder 2, but -5//3 has quotient -2 and remainder 1.
REGISTERS ALTERED:	R0,R1.
TASK SWITCHING:	Yes.
EXIT:	R0 contains the quotient. R1 contains the remainder.

Task switching means that the process may be suspended to run another process or accept a REPLY message. If task switching is not allowed, no other message at priority level 0 may be processed.

Appendix C. Utilities

The Utilities file defines data types and functions that are used throughout the compiler. The more important ones are listed below.

Bit Sets

The bset data type is defined in the Utilities file. A bset is an immutable abstract data type that represents a possibly infinite set of nonnegative integers. The operations allowed on bsets include testing for the empty bset, testing an integer for membership, adding and removing integers, finding unions, intersections, differences, and complements of bsets, returning the lowest integer present in the bset, and iterating using the loop facility and mapping over the members of bsets. Note that with these operations, the only possible bsets that can be created are either finite sets or complements of finite sets.

The Lisp reader is modified to accept a syntax for describing bsets. The syntax for a bset is

<bset> ::= #(|0|1){ (<integer> | (<integer> <integer>))* }

The digit following the # sign specifies whether the bset is a finite set or the complement of a finite set. If the digit is missing or 0, the bset is finite; otherwise, it is the complement of a finite set, and the nonnegative integers *not* in the set are listed. After the optional digit is the set of nonnegative integers in (or not in) the bset expressed as a list between braces. A range of consecutive integers may be specified by specifying a two-element list of the low and high integers, inclusive. Duplicate integers and overlapping ranges are allowed. Some examples of bsets are listed in Table C-1.

Table C-1. Sample Bsets

Syntax	Value
#{} or #0{}	The empty set {}
#1{}	The complete set {0,1,2,3,...}
#{4} or #{(4 4)}or #0{4 4 4 4 4 4}	The set {4}
#{1 2 8 9 3} or #{(1 3) 8 9}	The set {1,2,3,8,9}
#1{3 5}	The set {0,1,2,4,6,7,8,...}

Bsets are implemented as either integers or structures depending of whether *debug* is true. If *debug* is true, bsets are implemented as structures, which has the dual advantages of having bsets print in the readable format described above and type-checking operations on bsets, but at a penalty of increased garbage collection and about a 20% decrease in speed of the compiler.

FIFOs

The FIFO data type defined in Utilities is an implementation of a first-in-first-out queue. A FIFO can be created, elements can be added to the end of it, and the entire FIFO can be returned in the form of a list with the first elements listed first, all in constant time.

Digraphs

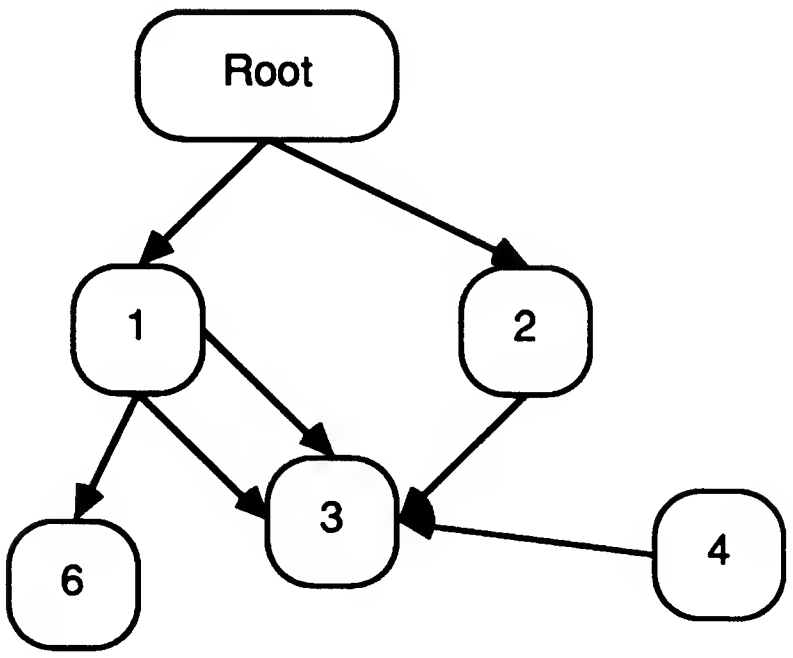


Figure C-1. Sample Digraph.
This digraph consists of a root and five nodes. Note that node 4 is not reachable with a depth-first traversal of the digraph originating at the root. Multiple edges between a pair of dinodes are permitted, as between nodes 1 and 3.

The Utilities file contains an extensive implementation of directed graphs (digraphs). A digraph consists of zero or more nodes (dinodes) and a root structure (the digraph structure itself). The root structure is linked to zero or more dinodes that are called "successors of the root." Each dinode is given a unique serial number to identify it during printing a digraph and debugging and to help in certain digraph operations. Besides the serial number, each dinode contains links to all of its predecessors and successors. In order for digraphs to be useful as an implementation of flow of control graphs for programs, all operations are careful to preserve the order of successors of each dinode. This way the "first" successor and "second" successor of conditional branch nodes are never transposed by digraph operations, which would reverse the meaning of the condition. Every digraph must be connected—any pieces not connected to the root are simply garbage-collected at the next opportunity. A sample digraph is shown in Figure C-1.

Dinodes and digraphs by themselves are structures that contain no user information. These structures are meant to be included in in other user-defined structures using Common Lisp's structure `:include` facility. This is the way stmts, stmtgraphs, insts, and modules are implemented.

Printing

The standard Common Lisp printer is inadequate in printing dinodes and digraphs because digraphs contain numerous circular references. Even with the depth and length limits and circular printing enabled, it is next to impossible to see the digraph structure from the Common Lisp printer's output. Thus, special printing procedures were defined for dinodes and digraphs.

A dinode is printed as a list of its data field names and values together with the lists of the serial numbers of the successors and predecessors of the dinode. If it is desired to see the values of the predecessor or successor dinodes, one can use the function nth-dinode to find in a digraph and output a dinode with a given serial number.

A digraph is printed in the format

#<digraph (<root1> ... <rootn>) {<set of dinode numbers>}>

The digraph is printed as a list of the serial numbers of the dinodes that are successors of the root followed by the serial numbers of all dinodes in the digraph printed in the set notation similar to that used by bmaps.

Low-Level Operations

The low-level operations on digraphs include creating an edge between two nodes, removing the edge, checking whether a dinode is the root, testing for an edge between two nodes, and traversing the digraph in depth-first order. Whenever the structure of a digraph is altered directly or with a low-level operation, altered-digraph should be called to cause some data structures pertaining to the digraph to be recalculated.

Medium-Level Operations

The medium-level operations on digraphs deal with the structure of more than just one or two nodes. These include inserting a new node in the place of an existing edge (Figure C-2a), inserting a new node before an existing one (Figure C-2b), deleting a node (Figure C-3), and merging two nodes into one (Figure C-4).

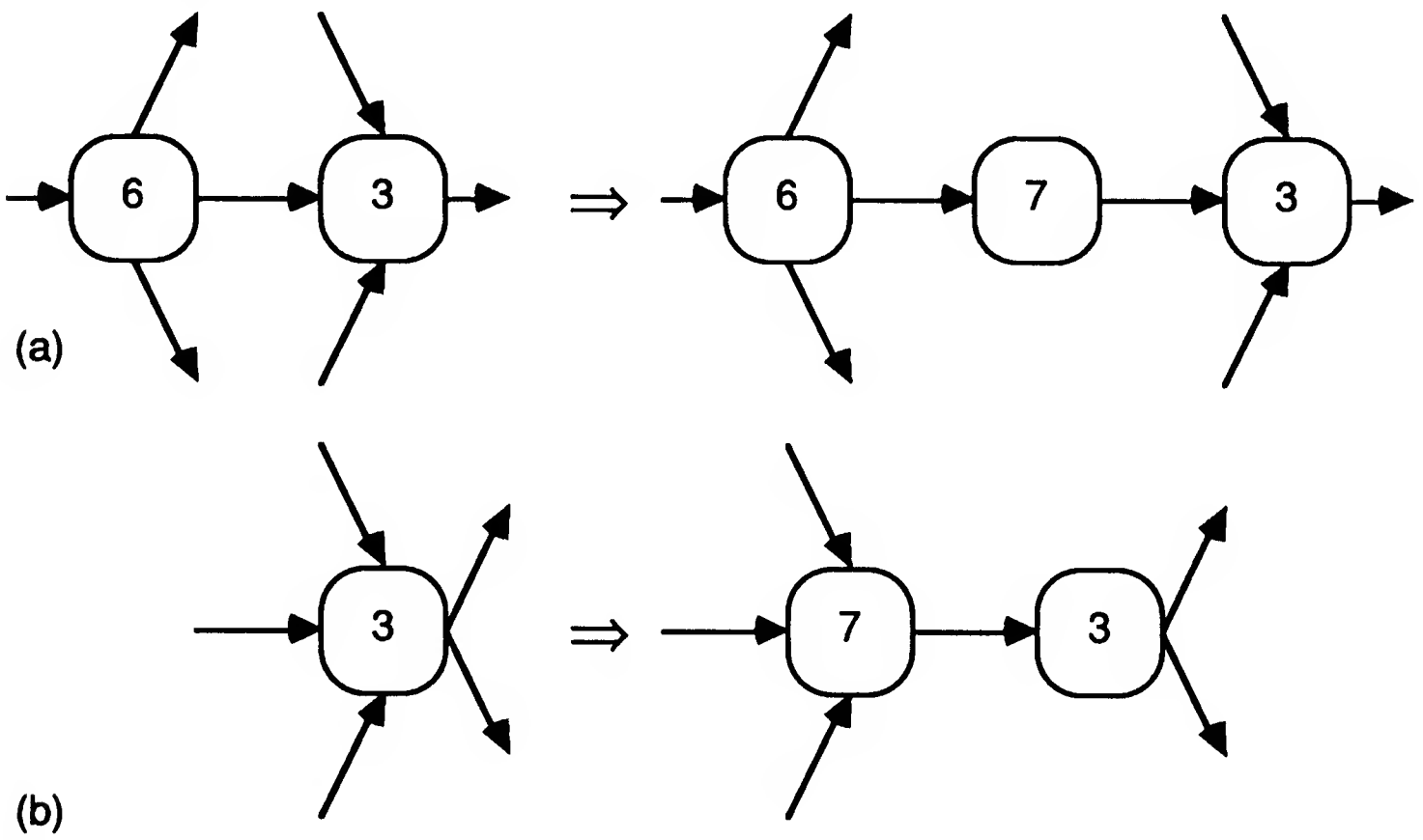


Figure C-2. Inserting a New Digraph Node.

(a) shows the effect of insert-dinode. A new dinode numbered 7 is inserted in place of an existing edge between existing dinodes 6 and 3.

(b) shows the effect of insert-before-dinode. The new dinode (7) is inserted before an existing one (3), and all of 3's predecessors are linked to the new dinode instead.

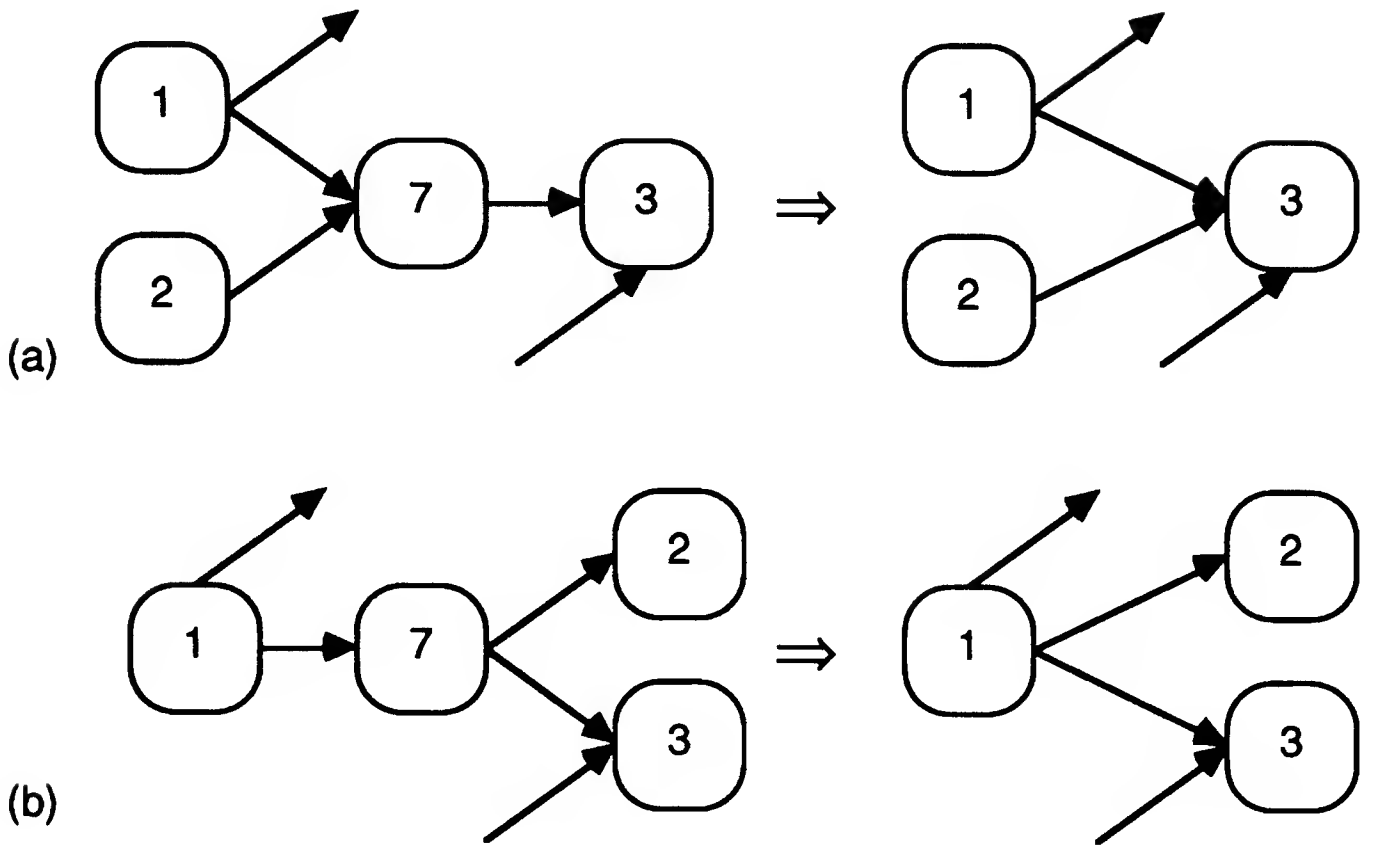


Figure C-3. Deleting a Digraph Node.
(a) and (b) both show the effect of calling delete-dinode on dinode 7. The connections between existing dinodes and the deleted dinode are transferred to the deleted dinode's predecessors and successors.

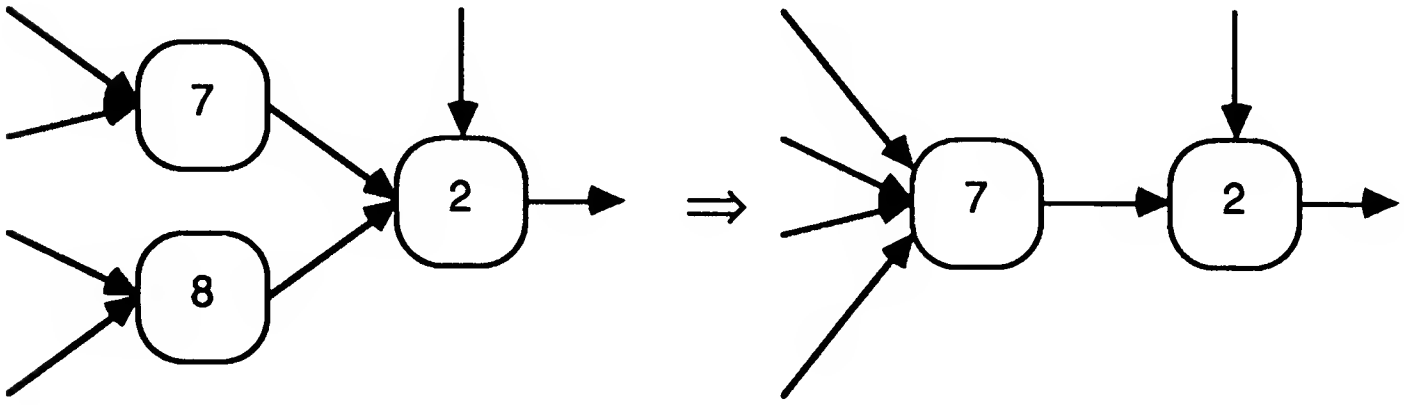


Figure C-4. Merging Digraph Nodes.
Dinodes 7 and 8 are merged by calling merge-dinode. All of 7's and 8's predecessors now connect to 7.

A predicate medium-level operation is also available that tests whether two dinodes connected by an edge are in the same basic block of the digraph. A basic block of a digraph is defined as a chain of nodes in which each node except the last has only one successor, the next node in the chain, and each node except the first has only one predecessor, the previous node in the chain.

Useful global medium-level operations include disconnecting all nodes that are not reachable by following edges from the root (Figure C-5), combining basic blocks into single nodes with a combinator function, and deleting all nodes that satisfy a given test from the digraph.

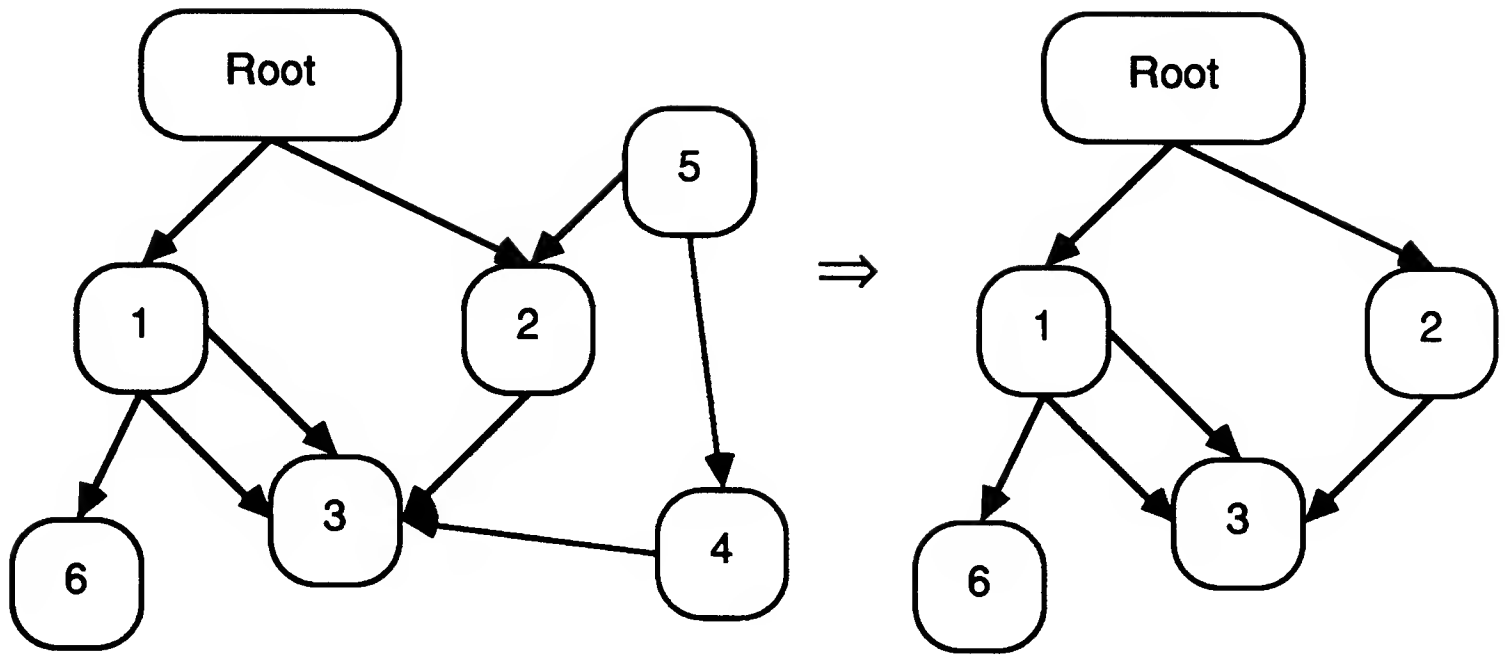


Figure C-5. Purging Unreachable Digraph Nodes.
Dinodes 4 and 5 are not reachable from the root of the digraph, so purge-unreachables-digraph disconnects them from the rest of the digraph. This operation is important for eliminating dead code and for ensuring consistency in algorithms that follow digraph edges in reverse.

High-Level Operations

The high-level operations on digraphs perform powerful macroscopic functions on the directed graphs. One such function, map-digraph, maps a digraph onto a new digraph by calling a mapping function for each node of the original digraph. The mapping function is allowed to return an arbitrary piece of a digraph, whether it be null, a single dinode, or many dinodes linked together.

Another high-level operation returns the digraph's blockgraph. A blockgraph is another digraph in which each node represents and points to an entire basic block of the original digraph.

The high-level digraph operation that is used the most is the general relaxation algorithm for digraphs. A routine, micro-relax, solves a relaxation problem of one of the forms:

Forward:
OutValue(root)=R
∀dinode d: InValue(d) = combinator(OutValue(pred1), ..., OutValue(predn)),
where pred1, ..., predn are d's predecessors
∀dinode d: OutValue(d) = f(InValue(d))
Backward:
∀dinode d: OutValue(d) = combinator(InValue(succ1), ..., InValue(succn)),
where succ1, ..., succn are d's successors
∀dinode d: InValue(d) = f(OutValue(d))

The algorithm used is similar to that described on page 691 of [2]. The algorithm proceeds by assigning an initial value to all of the dinodes. The initial values are important, as they determine the solution chosen when there are multiple solutions. It then iterates, calling f and combinator, over the entire digraph until the dinodes' values converge to a solution. The nodes' values are calculated in either the depth-first order or the reverse of the depth-first order, so as to maximize the speed of the convergence. It can be shown that for the functions f and combinator used in the compiler ($f(x) = (x \wedge a) \vee b$, combinator is either $\wedge$ or $\vee$ of the arguments), the relaxation algorithm will always converge to a solution.

Relax is a variant of micro-relax that first computes the blockgraph, performs the iterative algorithm on the blockgraph, and then calculates the values for the original digraph. Relax may be faster than micro-relax on digraphs containing many loops.

Calc-dominators calculates the *dominators* of each dinode. A node A is a dominator of node B≠A if every path from the root to B must pass through A.

Finally, linearize is a function that returns a list of the nodes of the digraph in an order that attempts to minimize the number and length of *branches*, where a branch is an edge from a dinode to a dinode that does not follow it in the list that is output. Linearize uses heuristics such as listing all of the dominators of a dinode before the dinode and keeping track of the loops encountered in the digraph and, when a loop has been entered, assigning priority to listing dinodes in the loops before dinodes outside the loops.

Appendix D. Complete Listing of the Compiler

Utilities

```
;;;
```

```
            ((/= num (1+ last))
             (print-rangeend rangebegin last)
             (write-char #\Space stream)
             (setq rangebegin num)))
        (setq last num)))
    (write-char #\) stream)
    (values)))


;;;Get the value of an attribute from an association list.  If it is not present,
;;;calculate it using the calculator expression and add it to the association list.
;;;Return the value of the attribute.
(defmacro attribute (name assoc-list calculator)
  (let ((value (gensym))
        (new-value (gensym)))
    `(let ((,value (assoc ,name ,assoc-list)))
       (if ,value
         (cdr ,value)
         (let ((,new-value ,calculator))
           (push (cons ,name ,new-value) ,assoc-list)
           ,new-value)))))

;;;Perform action until it does not clear the attributes.
(defmacro attribute-steady-state (assoc-list action)
  (let ((name (gensym)))
    `(loop do (progn
                (unless (assoc ',name ,assoc-list) (push '(,name) ,assoc-list))
                ,action)
           until (assoc ',name ,assoc-list))))


;;;Return the floor of the base-2 logarithm of positive integer n.
;;;Return nil if n is zero or negative.
(defun log2 (n)
  (and (> n 0) (1- (integer-length n))))

;;;Return true if integer n is a power of 2.
(defun power-of-2? (n)
  (and (> n 0) (eql n (ash 1 (log2 n)))))

;;;;+--------+
;;;;| BitMap |
;;;;+--------+
;;;A bmap is a bitmap of the variables, implemented as an integer or a structure depending
;;;on whether *debug* is true.
;;;The following operations are defined:

(eval-when (compile load eval)
  (if *debug*
    (progn
      (fresh-line)
      (write-line ";bmap is a structure.")
      (defstruct (bmap (:print-function print-bmap)) bits)
      (defmacro int-to-bmap (i) `(make-bmap :bits ,i))
      (defmacro bmap-to-int (b) `(bmap-bits ,b)))

    (progn
      (deftype bmap () 'integer)
      (defmacro int-to-bmap (i) i)
      (defmacro bmap-to-int (b) b))))


(defconstant b0 (int-to-bmap 0)) ;The empty bset.
(defconstant b1 (int-to-bmap -1)) ;The complete bset.

;;;Return true if the bmap is empty.
(defmacro bempty (bmap)
  `(zerop (bmap-to-int ,bmap)))

;;;Make a bmap with bits from low-bit (inclusive) to high-bit (exclusive) set.
;;;Low-bit defaults to zero.
(defmacro brange (high-bit &optional low-bit)
  (list 'int-to-bmap
        (if low-bit `(- (1- (ash 1 ,high-bit)) (1- (ash 1 ,low-bit)))
            `(1- (ash 1 ,high-bit)))))

;;;Return non-nil if variable n is in the bmap.
(defmacro btest (n bmap)
  `(logbitp ,n (bmap-to-int ,bmap)))

;;;Add variable n to the bmap.  If bmap is missing, return a bmap with only
```

```
;;;variable n set.
(defmacro bset (n &optional bmap)
  (list 'int-to-bmap
        (if bmap
          `(logior (ash 1 ,n) (bmap-to-int ,bmap))
          `(ash 1 ,n))))
(define-modify-macro bsetf (n) (lambda (bmap n) (bset n bmap)))

;;;Remove variable n from the bmap.
(defmacro bclr (n bmap)
  `(int-to-bmap (logandc1 (ash 1 ,n) (bmap-to-int ,bmap))))
(define-modify-macro bclrf (n) (lambda (bmap n) (bclr n bmap)))

;;;Return the lowest-numbered variable greater than or equal to low in the bmap
;;;or nil if the bmap is empty.
(defun blow (bmap &optional (low 0))
  (cond ((and (>= (bmap-to-int bmap) 0) (< (bmap-to-int bmap) (ash 1 low))) nil)
        ((btest low bmap) low)
        (t (blow bmap (1+ low)))))

;;;Return the union of the bmaps.
(defmacro b+ (&rest bmaps)
  (list 'int-to-bmap
        (cons 'logior
              (mapcar #'(lambda (bmap) (list 'bmap-to-int bmap)) bmaps))))

;;;Return the intersection of the bmaps.
(defmacro b* (&rest bmaps)
  (list 'int-to-bmap
        (cons 'logand
              (mapcar #'(lambda (bmap) (list 'bmap-to-int bmap)) bmaps))))

;;;Return the difference of two bmaps.
(defmacro b- (bmap1 bmap2)
  `(int-to-bmap (logandc2 (bmap-to-int ,bmap1) (bmap-to-int ,bmap2))))

;;;Return the complement of the bmap.
(defmacro bnot (bmap)
  `(int-to-bmap (lognot (bmap-to-int ,bmap))))

;;;Return the union of the results of the function f applied to the elements of the list l.
(defun map-b+ (f l)
  (do ((lst l (cdr lst))
       (result b0 (b+ (funcall f (car lst)) result)))
      ((endp lst) result)))

;;;Return the intersection of the results of the function f applied to the elements of the list l.
(defun map-b* (f l)
  (do ((lst l (cdr lst))
       (result b1 (b* (funcall f (car lst)) result)))
      ((endp lst) result)))


;;;Add mapping over variables of a bmap to the loop macro.
;;;The format is:
;;; (loop for <bit-var> being the bits of <bmap> ...)
;;;<bit-var> gets assigned to each set bit of <bmap>.  The order is not specified.
(define-loop-path bits bits-path (of))
(defun bits-path (path-name variable data-type prep-phrases inclusive? allowed-prepositions data)
  (declare (ignore path-name data-type allowed-prepositions data))
  (let ((of-phrase (loop-tassoc 'of prep-phrases))
        (bmap (gensym)))
    (cond
     ((null of-phrase) (error "OF missing"))
     (inclusive? (error "Inclusive iteration path not supported")))
    (list
     (list (list variable -1)
           (cons bmap (cdr of-phrase)))
     nil
     nil
     (list variable `(blow ,bmap (1+ ,variable)))
     `(null ,variable)
     nil)))


(eval-when (compile load eval)
  (when *debug*
    (defun print-bmap (bmap stream depth)
      (declare (ignore depth))
      (write-char #\# stream)
```

```
      (let ((bits (bmap-to-int bmap)))
        (when (< bits 0)
          (write-char #\1 stream)
          (setq bits (lognot bits)))
        (print-range-list #'identity
                          (loop for i being the bits of (int-to-bmap bits) collect i)
                          stream)))))


;;;Allow printed bmaps to be read back in by the reader.
(eval-when (compile load eval)
  (defun bmap-reader (stream char arg)
    (declare (ignore char))
    (let ((bits b0))
      (dolist (num (read-delimited-list #\} stream t))
        (cond
         ((integerp num) (bsetf bits num))
         ((consp num)
          (let ((n1 (first num))
                (n2 (second num)))
            (if (and (integerp n1)
                     (integerp n2)
                     (null (cddr num)))
              (setq bits (b+ bits (if (>= n1 n2) (brange (1+ n1) n2)
                                                 (brange (1+ n2) n1))))
              (error "Bad bmap range: ~S" num))))
         (t (error "Bad bmap bit: ~S" num))))
      (if (and arg (> arg 0))
        (setq bits (bnot bits)))
      bits))

  (set-dispatch-macro-character #\# #\{ #'bmap-reader)
  (set-macro-character #\)
                       #'(lambda (stream char)
                           (declare (ignore stream char))
                           (error "Bad bmap specification"))
                       nil))


;;;;+------+
;;;;| FIFO |
(deftype fifo () 'cons)

;;;Create a new FIFO queue from the given data list.
(defun new-fifo (&optional data)
  (if (null data)
    (let ((n (cons nil nil)))
      (setf (car n) n)
      n)
    (cons (last data) data)))

;;;Return the fifo's data as a list.
(defmacro fifo-data (fifo)
  `(cdr ,fifo))

;;;Return the fifo's last element or nil if there isn't any.
(defun fifo-tail (fifo)
  (if (eq (car fifo) fifo)
    nil
    (caar fifo)))

;;;Add an element to the end of a fifo.
(defun add-fifo (fifo element)
  (let ((pair (cons element nil)))
    (setf (cdar fifo) pair)
    (setf (car fifo) pair)
    fifo))


;;;;+----------------+
;;;;| A digraph node |
;;;;+----------------+
;;;(dinode-predecessors dinode) is a list of the dinode's predecessors.  The order is not
;;; important.  A nil predecessor indicates the head of the digraph.
;;;(dinode-successors dinode) is a list of the dinode's successors.  The order is
;;; important--for conditional branches, the first successor is the fall-through
;;; destination, while the second successor is the branch destination.
;;;(dinode-serial-number dinode) is a serial number of the dinode for debugging purposes.
;;;(dinode-mark dinode) is a mark temporary used for searching the digraph.
;;;(dinode-spare dinode) is a spare value used by map-digraph.
```

```
;;;There is no data field; the dinodes are meant to be used as part of larger records with
;;;other data fields.
(defvar current-serial-number 0)
(defstruct (dinode (:print-function print-dinode))
  (serial-number (setq current-serial-number (1+ current-serial-number)) :read-only t)
  mark
  spare
  predecessors
  successors)


;;;Return a list representation of the dinode's data.
(defun printdata-dinode (dinode)
  (list
   (dinode-serial-number dinode)
   (cons 'predecessors
         (mapcar #'dinode-serial-number (dinode-predecessors dinode)))
   (cons 'successors
         (mapcar #'dinode-serial-number (dinode-successors dinode)))))

;;;Print the dinode.
(defun print-dinode (dinode stream depth)
  (declare (ignore depth))
  (prinl (cons 'dinode (printdata-dinode dinode)) stream))


;;;Replace the first occurence of old in data with new-list spliced into data.
;;;A new data is returned; the alteration is nondestructive.
;;;An error is given if old is not found in data.
(defun subst-append-1 (data old new-list)
  (cond
   ((null data) (error "Old not found in data in subst-append-1"))
   ((eq (car data) old) (append new-list (cdr data)))
   (t (cons (car data) (subst-append-1 (cdr data) old new-list)))))
;;;(Subst-append-1f data old new-list) sets the generalized variable data to be the
;;;(subst-append-1 data old new-list).
(define-modify-macro subst-append-1f (old new-list) subst-append-1)


;;;Link the from dinode to the to dinode.
;;;No check is made for duplicate links.
(defun link-dinode (from to)
  (setf (dinode-successors from) (nconc (dinode-successors from) (list to)))
  (push from (dinode-predecessors to)))


;;;Unlink the from dinode from the to dinode.  The link must have been present.
;;;If there was more than one link from the from dinode to the to dinode, only one
;;;link is removed.
(defun unlink-dinode (from to)
  (subst-append-1f (dinode-successors from) to '())
  (subst-append-1f (dinode-predecessors to) from '()))


;;;Return true if the from dinode is linked to the to dinode.
(defun dinodes-linked-p (from to)
  (find to (dinode-successors from)))


;;;;+-----------+
;;;;| A digraph |
;;;;+-----------+
;;;A digraph is a collection of dinodes with a digraph header.
;;;(digraph-successors digraph) is the list of the head dinodes of the digraph.
;;;(digraph-mark digraph) is an integer such that none of the dinodes in the digraph has
;;; a mark greater than to (digraph-mark digraph).
;;;(digraph-attributes digraph) is an association list of information about the digraph
;;; that is cleared every time the digraph is altered.  The list includes:
;;; (digraph-dflist <list>), a depth-first ordered list of the digraph's nodes;
;;; (digraph-reverse-dflist <list>), the reverse of digraph-dflist;
;;; as well as other user-defined items.
(defstruct (digraph (:include dinode (serial-number nil) (mark 0))
                    (:print-function print-digraph))
  attributes
  dflist
  reverse-dflist)


;;;Return a list representation of the digraph's data.
;(defun printdata-digraph (digraph)
;  (list (mapcar #'dinode-serial-number (digraph-successors digraph))
```

```
;        (mapcar #'dinode-serial-number (digraph-dfs digraph))))

;;;Print the digraph.
;(defun print-digraph (digraph stream depth)
;  (declare (ignore depth))
;  (prin1 (cons 'digraph (printdata-digraph digraph)) stream))


(defun print-digraph-data (digraph stream)
  (format stream "~S " (mapcar #'dinode-serial-number (digraph-successors digraph)))
  (print-range-list #'dinode-serial-number (digraph-dfs digraph) stream))

;;;Print the digraph.
(defun print-digraph (digraph stream depth)
  (declare (ignore depth))
  (write-string "#<Digraph " stream)
  (print-digraph-data digraph stream)
  (write-char #\> stream))


;;;Return true if the dinode is actually the root of the digraph.
(defmacro root? (dinode)
  `(null (dinode-serial-number ,dinode)))
(defmacro non-root? (dinode)
  `(dinode-serial-number ,dinode))


;;;Create a new digraph with the given dinodes as successors of the root.
;;;If root is non-nil, it is used as the root of the digraph; otherwise, a new
;;;root is created.
(defun new-digraph (root-successors &optional root)
  (let ((graph (or root (make-digraph))))
    (dolist (successor root-successors graph)
      (link-dinode graph successor))))


;;;Mark the digraph as altered.
(defun altered-digraph (digraph)
  (setf (digraph-attributes digraph) nil))
```

```
;;;Return a depth-first ordered list of the digraph's nodes.
;;;If :from-end is non-nil, return the list in reverse order.
(defun digraph-dfs (digraph &key from-end)
  (if from-end
    (attribute
     'reverse-dflist (digraph-attributes digraph)
     (reverse (digraph-dfs digraph)))
    (attribute
     'dflist (digraph-attributes digraph)
     (let ((mark (1+ (digraph-mark digraph))))
       (labels
         ((dfs (dinode)
           (unless (eql (dinode-mark dinode) mark)
             (setf (dinode-mark dinode) mark)
             (cons dinode (mapcan #'dfs (dinode-successors dinode))))))
         (cdr (dfs digraph)))))))


;;;Call function f on each dinode in the digraph in depth-first order.
;;;If from-end is non-nil, call function f in the reverse of the depth-first order.
;;;If order is non-nil, call function f on the nodes present in the order list.
(defun all-dinodes (digraph f &key from-end)
  (mapc f (digraph-dfs digraph :from-end from-end)))


;;;Add mapping over digraphs to the loop macro.
;;;The format is:
;;; (loop for <node-var> being the dinodes of <digraph> [:from-end <bool-expr>] ...)
;;;<node-var> gets assigned to each dinode of <digraph> in depth-first order
;;;or the reverse of the depth-first order if :from-end is specified and has a non-nil value.
(define-loop-path dinodes dinodes-path (of :from-end))
(defun dinodes-path (path-name variable data-type prep-phrases inclusive? allowed-prepositions data)
  (declare (ignore path-name data-type allowed-prepositions data))
  (let ((of-phrase (loop-tassoc 'of prep-phrases))
        (from-end-phrase (loop-tassoc :from-end prep-phrases))
        (cursor (gensym)))
    (cond
     ((null of-phrase) (error "OF missing"))
     (inclusive? (error "Inclusive iteration path not supported")))
    (list
```

```
     (list (list variable)
           (list cursor `(digraph-dfs ,(cadr of-phrase) :from-end ,(cadr from-end-phrase))))
     nil
     `(null ,cursor)
     nil
     nil
     (list variable `(car ,cursor)
           cursor `(cdr ,cursor)))))


;;;Return the node in the digraph with serial number n, if any.
;;;This function is intended for debugging.
(defun nth-dinode (digraph n)
  (find n (digraph-dfs digraph) :key #'dinode-serial-number))


;;;Purge any unreachable dinodes from the digraph.
;;;This routine works by examining the marks left from the last digraph-dfs; hence, the
;;;marks or the digraph should not be modified since the last call to digraph-dfs.
(defun purge-unreachables-digraph (digraph)
  (loop for dinode being the dinodes of digraph do
        (setf (dinode-predecessors dinode)
              (delete-if #'(lambda (predecessor)
                             (and predecessor (not (eql (dinode-mark predecessor)
                                                        (digraph-mark digraph)))))
                         (dinode-predecessors dinode))))
  digraph)


;;;Check if the dinode is linked to itself.  If so, remove all such links and return true;
;;;otherwise, return false.
(defun unlink-self-dinode (dinode)
  (when (dinodes-linked-p dinode dinode)
    (unlink-dinode dinode dinode)
    (unlink-self-dinode dinode)
    t))


;;;Destructively delete a dinode from a digraph, connecting its predecessors
;;;to its successors.
(defun delete-dinode (dinode)
  (unlink-self-dinode dinode)
  (dolist (predecessor (dinode-predecessors dinode))
    (subst-append-1f (dinode-successors predecessor) dinode (dinode-successors dinode)))
  (dolist (successor (dinode-successors dinode))
    (subst-append-1f (dinode-predecessors successor) dinode (dinode-predecessors dinode)))
  (setf (dinode-predecessors dinode) nil)
  (setf (dinode-successors dinode) nil))


;;;Insert new-dinode between pred-dinode and succ-dinode, breaking the link between
;;;pred-dinode and succ-dinode.
(defun insert-dinode (new-dinode pred-dinode succ-dinode)
  (subst-append-1f (dinode-successors pred-dinode) succ-dinode (list new-dinode))
  (subst-append-1f (dinode-predecessors succ-dinode) pred-dinode (list new-dinode))
  (setf (dinode-predecessors new-dinode) (list pred-dinode))
  (setf (dinode-successors new-dinode) (list succ-dinode)))


;;;Insert new-dinode in front of dinode, linking new-dinode with all of dinode's predecessors
;;;and dinode as the successor.
(defun insert-before-dinode (new-dinode dinode)
  (dolist (predecessor (dinode-predecessors dinode))
    (subst-append-1f (dinode-successors predecessor) dinode (list new-dinode)))
  (setf (dinode-predecessors new-dinode) (dinode-predecessors dinode))
  (setf (dinode-successors new-dinode) (list dinode))
  (setf (dinode-predecessors dinode) (list new-dinode)))


;;;Merge dinode1 with dinode2.  The successor of the result is dinode1's successor.  The
;;;predecessors are the union of the two dinodes' predecessors.
(defun merge-dinodes (dinode1 dinode2)
  (dolist (successor (dinode-successors dinode2))
    (unlink-dinode dinode2 successor))
  (dolist (predecessor (dinode-predecessors dinode2))
    (subst-append-1f (dinode-successors predecessor) dinode2 (list dinode1)))
  (setf (dinode-predecessors dinode1)
        (append (dinode-predecessors dinode1) (dinode-predecessors dinode2)))
  (setf (dinode-predecessors dinode2) nil))
```

```
;;;Destructively concatenate sequences of dinodes in the digraph using the concatenator
;;;function to combine the dinodes.  Return the digraph.
;;;The combinator function must alter the structure of its first dinode argument to combine
;;;it with the second dinode.
(defun combine-digraph (digraph combinator)
  (loop for dinode being the dinodes of digraph do
        (let ((predecessor (car (dinode-predecessors dinode))))
          (when (and predecessor
                     (one-elt-p (dinode-predecessors dinode))
                     (one-elt-p (dinode-successors predecessor)))
            (funcall combinator predecessor dinode)
            (delete-dinode dinode)
            (altered-digraph digraph))))
  digraph)


;;;Destructively remove empty dinodes from the digraph, collapsing the structure of the
;;;digraph as nodes are removed.  The empty function decides whether a dinode is empty.
;;;Nodes with more than one successor should not be flagged as empty.
;;;Return the digraph.
(defun delete-dinode-if (digraph empty)
  (loop for dinode being the dinodes of digraph do
        (when (funcall empty dinode)
          (delete-dinode dinode)
          (altered-digraph digraph)))
  digraph)


;;;Create a new digraph which is a mapping of the current one.  The mapping function map,
;;;passed as a parameter, takes a single argument which is a node of the current digraph.
;;;It returns two values; the first one is a starting dinode, while the second one is a
;;;list of ending dinodes.  Map-digraph maps each dinode in the original digraph into a
;;;new digraph with the mapping function determining what each dinode maps into.
;;;Map may graph a dinode into nothing by returning two nils.  Two restrictions apply when
;;;it does this:
;;; No loops of empty nodes may be created.
;;; A dinode may not map into nothing if it has more than one successor.
;;;Return the new digraph.
;;;If order is present, it must be a list of the digraph's nodes.  Map is called in the
;;;order specified by order.
(defun map-digraph (digraph map &key order)
  ;;First find the mapping of each dinode and store it in spare.  If the dinode maps into
  ;;nothing, replace it with a dummy dinode indicated by a t in its spare variable.
  (dolist (dinode (or order (digraph-dfs digraph)))
    (multiple-value-bind (first last) (funcall map dinode)
      (if first
        (setf (dinode-spare dinode) (cons first last))
        (let ((dummy-dinode (make-dinode :spare t)))
          (setf (dinode-spare dinode) (list dummy-dinode dummy-dinode))))))
  ;;Now join the mapped dinodes to each other.
  (loop for dinode being the dinodes of digraph do
        (dolist (successor (dinode-successors dinode))
          (dolist (final-node (cdr (dinode-spare dinode)))
            (link-dinode final-node (car (dinode-spare successor))))))
  ;;Finally initialize the new digraph and remove the dummy mapped dinodes.
  (let ((newgraph (make-digraph)))
    (dolist (successor (digraph-successors digraph))
      (link-dinode newgraph (car (dinode-spare successor))))
    (delete-dinode-if
     newgraph
     #'dinode-spare)))


;;;Return true if successor and predecessor are nodes in the same digraph with a
;;;link from predecessor to successor and with no other links from predecessor or
;;;to successor and with predecessor not being the root.
(defun in-same-basic-block? (predecessor successor)
  (and (non-root? predecessor)
       (one-elt-p (dinode-predecessors successor))
       (one-elt-p (dinode-successors predecessor))
       (eq (car (dinode-successors predecessor)) successor)))


;;;;+------------+
;;;;| BlockGraph |
;;;;+------------+
;;;A block graph is a digraph in which each node represents a block of nodes in another
;;;digraph.  A block of nodes is a string of one or more nodes in which every node except
;;;the last has as its only successor the next node in the string and every node except the
;;;first has as its only predecessor the previous node in the string.
```

```
;;;(block-nodes block) is a list containing this block's string of nodes.
;;;(block-reverse-nodes block) is the reverse of (block-nodes block).
;;;(block-set block), (block-clear block) and  (block-val block)
;;; are temporary variables for the relaxation algorithm.
;;;(block-number block) is this block's number used for the dominator algorithm.
;;;(block-dominators block) is a bmap of the block numbers of the dominators of this block.
(defstruct (block (:include dinode) (:print-function print-block))
  nodes
  reverse-nodes
  set
  clear
  val
  number
  dominators)

;;;Return a list representation of the block's data.
(defun printdata-block (block)
  (nconc (printdata-dinode block)
         (list
          (cons 'nodes (mapcar #'dinode-serial-number (block-nodes block)))
          (list 'set (block-set block))
          (list 'clear (block-clear block))
          (list 'val (block-val block))
          (list 'number (block-number block))
          (list 'dominators (block-dominators block)))))

;;;Print the block.
(defun print-block (block stream depth)
  (declare (ignore depth))
  (prin1 (cons 'block (printdata-block block)) stream))


;;;Get the block graph of a digraph.  If an existing block graph is available and the digraph
;;;hasn't been changed since it was made, use the existing block graph; otherwise, create a
;;;new one.
(defun get-blockgraph (digraph)
  (attribute
   'blockgraph (digraph-attributes digraph)
   (let ((blockgraph (make-digraph)))
     (loop for dinode being the dinodes of digraph do (setf (dinode-spare dinode) nil))
     (setf (dinode-spare digraph) blockgraph)
     (loop for dinode being the dinodes of digraph do
           (let ((predecessor (car (dinode-predecessors dinode))))
             (if (in-same-basic-block? predecessor dinode)
                 (progn
                   (setf (dinode-spare dinode) (dinode-spare predecessor))
                   (push dinode (block-reverse-nodes (dinode-spare dinode))))
                 (progn
                   (setf (dinode-spare dinode)
                         (make-block :reverse-nodes (list dinode)))
                   (dolist (predecessor (dinode-predecessors dinode))
                     (if (dinode-spare predecessor)
                         (link-dinode (dinode-spare predecessor) (dinode-spare dinode))))))))
           (dolist (successor (dinode-successors dinode))
             (if (dinode-spare successor)
                 (link-dinode (dinode-spare dinode) (dinode-spare successor))))))
     ;;Calculate the nodes variables.
     (loop for block being the dinodes of blockgraph do
           (setf (block-nodes block)
                 (reverse (block-reverse-nodes block))))
     blockgraph)))


;;;;+------------------------------+
;;;;| Dataflow Relaxation Algorithm |
;;;;+------------------------------+

;;;Calculate the equilibrium values for each node of the digraph that satisfy the
;;;identity
;;; (combinator (list (get-result predecessor1) (get-result predecessor2)...))=(get-val).
;;;The algorithm works by assigning initial-val to each node and then recalculating the
;;;values for each node using the above equation until an equilibrium is reached.
;;;An equilibrium is defined by the above equation holding with #'equalp as a test.
;;;
;;;The parameter functions are as follows:
;;;(get-val dinode) returns the node's current input value.
;;;(set-val dinode val) sets the node's input value to val.
;;;(get-result dinode) returns the node's output value calculated from the current input value.
;;; It should return the root value if called on the root.
;;;initial-val is the initial input value for all nodes.
;;;root-val is the root's result value.
```

```
;;;(combinator extractor list) is a function that combines the values returned by the extractor
;;; function for each predecessor in the list.
;;;from-end specifies whether the node's input comes from its predecessors (nil) or
;;; successors (non-nil).  If from-end is true, the relaxation proceeds backwards, from
;;; the successors to the predecessors.
(defun macro-relax (digraph get-val set-val get-result
                            &key (initial-val b0) (root-val b0)
                            (combinator #'map-b+)
                            from-end)
  ;;Clear the values.
  (loop for node being the dinodes of digraph do (funcall set-val node initial-val))
  ;;Now iterate through the digraph until no more changes occur.
  (loop for changed = nil do
        (loop for node being the dinodes of digraph :from-end from-end do
              (let ((new-val
                     (funcall combinator
                              #'(lambda (dinode)
                                  (if (root? dinode) root-val
                                      (funcall get-result dinode)))
                              (if from-end (dinode-successors node)
                                  (dinode-predecessors node)))))
                (unless (equalp (funcall get-val node) new-val)
                  (funcall set-val node new-val)
                  (setq changed t))))
        while changed)
  ;;Evaluate get-result on all digraph nodes in case it has any side effects.
  (if from-end
    (dolist (root-successor (digraph-successors digraph))
      (funcall get-result root-successor)))
  digraph)


;;;Calculate the equilibrium values for each node of the digraph that satisfy the
;;;identity
;;; (combinator (list (result predecessor1) (result predecessor2)...))=(get-val),
;;;where the result function is defined as
;;; (result input):=(union set (difference input clear)).
;;;The algorithm works by conceptually assigning initial-val to each node and then
;;;recalculating the values for each node using the above equation until an equilibrium
;;;is reached.  An equilibrium is defined by the above equation holding with #'equalp as
;;;a test.  Actually, the blocks in the digraph are located and the iteration proceeds only
;;;on the whole blocks and is later distributed to the entire digraph.  For this reason
;;;it is important that union and difference behave like ordinary set union and difference.
;;;
;;;The parameter functions are as follows:
;;;(get-val dinode) returns the node's current input value.
;;;(set-val dinode val) sets the node's input value to val.
;;;(get-set dinode) returns the node's set value.
;;;(get-clear dinode) returns the node's clear value.
;;;root-val is the root's result value.
;;;(combinator extractor list) is a function that combines the values returned by the extractor
;;; function for each predecessor in the list.
;;;from-end specifies whether the node's input comes from its predecessors (nil) or
;;; successors (non-nil).  If from-end is true, the relaxation proceeds backwards, from
;;; the successors to the predecessors.
(defun relax (digraph get-val set-val get-set get-clear
                      &key (initial-val b0) (root-val b0)
                      (union #'logior) (difference #'logandc2) (combinator #'map-b+)
                      from-end)
  (declare (ignore get-val))
  (labels

    ;;Calculate the set and clear values in the block.
    ((calc-set-clear (block)
      (let ((ns
             (if from-end ;Do the calculation backwards to minimize get-clear calls.
               (block-nodes block)
               (block-reverse-nodes block))))
        (do ((nodes (cdr ns) (cdr nodes))
             (set (funcall get-set (car ns))
                  (funcall union set
                           (funcall difference (funcall get-set (car nodes)) clear)))
             (clear (funcall get-clear (car ns))
                    (funcall union clear (funcall get-clear (car nodes)))))
            ((endp nodes)
             (setf (block-set block) set)
             (setf (block-clear block) clear)))))

     ;;Calculate the information inside a block from the information on its boundaries.
     (micro-relax (block)
      (do ((nodes (if from-end
```

```
                    (block-reverse-nodes block)
                    (block-nodes block))
                  (cdr nodes))
            (val (block-val block)
                 (funcall union (funcall get-set (car nodes))
                          (funcall difference val (funcall get-clear (car nodes))))))
          ((endp nodes))
        (funcall set-val (car nodes) val))))

    (let ((blockgraph (get-blockgraph digraph)))
      (all-dinodes blockgraph #'calc-set-clear)
      (macro-relax
       blockgraph
       #'block-val
       #'(lambda (block val) (setf (block-val block) val))
       #'(lambda (block)
           (funcall union (block-set block)
                    (funcall difference (block-val block) (block-clear block))))
       :initial-val initial-val
       :root-val root-val
       :combinator combinator
       :from-end from-end)
      (all-dinodes blockgraph #'micro-relax))
    digraph))


;;;;+--------------------+
;;;;| Dominator Algorithm |
;;;;+--------------------+

;;;Number the blocks in the blockgraph consecutively starting at 1.
(defun number-blocks (blockgraph)
  (loop for n from 1
        for block being the dinodes of blockgraph do
        (setf (block-number block) n)))

;;;Calculate the set of dominators for each block in the digraph.  No block dominates itself.
;;;The dominators of each block are stored in the block's (block-dominators) location as a
;;;bset of block numbers.
;;;Return the digraph's blockgraph.
(defun calc-dominators (digraph)
  (number-blocks (get-blockgraph digraph))
  (macro-relax
   (get-blockgraph digraph)
   #'block-dominators
   #'(lambda (block val) (setf (block-dominators block) val))
   #'(lambda (block) (bset (block-number block) (block-dominators block)))
   :initial-val b1
   :root-val b0
   :combinator #'map-b*))

;;;Return true if block1 dominates block2.  Calc-dominators should have been run before
;;;this routine is called.
(defun dominatesp (block1 block2)
  (btest (block-number block1) (block-dominators block2)))


;;;;+--------------------+
;;;;| Linearize a digraph |
;;;;+--------------------+

;;;Order the elements of lst according to the priorities specified in priorities.
;;;The returned list consists of the elements of lst ordered by which ones satisfy
;;;the elements of priorities--the elements of lst that satisfy (first priorities)
;;;are listed first, the elements of lst that satisfy (second priorities) are listed
;;;next, and so on, until the elements of lst that don't satisfy any element of
;;;priorities, which are listed last.
(defun priority-order (lst priorities satisfied)
  (cond
   ((endp priorities) lst)
   ((endp lst) nil)
   ((endp (cdr lst)) lst)
   ((find (car priorities) lst :test satisfied)
    (cons (car priorities)
          (priority-order (remove (car priorities) lst
                                  :test satisfied
                                  :count 1) priorities satisfied)))
   (t (priority-order lst (cdr priorities) satisfied))))

;;;Order the blocks in the blockgraph in a linear fashion.
(defun order-blocks (blockgraph)
```

```
  (let ((ordering (new-fifo)))
    (labels
      ((order (block priority)
        (setf (block-val block) t)
        (add-fifo ordering block)
        (dolist (successor (priority-order (dinode-successors block)
                                           priority
                                           #'(lambda (priority successor)
                                               (or (eq priority successor)
                                                   (dominatesp successor priority)))))
          (unless (block-val successor)
            ;;Order a block iff all of its predecessors have been ordered, are
            ;;equal to that block, or are dominated by that block.
            (let ((new-priority priority))
              (if (every
                   #'(lambda (predecessor)
                       (or (root? predecessor)
                           (block-val predecessor)
                           (eq predecessor successor)
                           (progn
                             (push predecessor new-priority)
                             (dominatesp successor predecessor))))
                   (dinode-predecessors successor))
                  (order successor new-priority)))))))

       ;;Use the (block-val) variables to mark nodes that have already been ordered.
       (loop for block being the dinodes of blockgraph do (setf (block-val block) nil))
       (loop for block being the dinodes of blockgraph do
             (unless (block-val block) (order block nil)))
       (fifo-data ordering))))

;;;Linearize the digraph--order the blocks in a way that attempts to minimize
;;;the number of branches.
;;;Return a list containing all nodes of the graph in the preferred order.
(defun linearize (digraph)
  (mapcan #'(lambda (block) (copy-list (block-nodes block)))
          (order-blocks (calc-dominators digraph))))
```

Word

```
;;;
```

```
;;;Short words:
(defconstant short-words (list wNIL
                               wFALSE
                               wTRUE
                               (make-word #x-80000000)
                               (make-word #x000000FF)
                               (make-word #x000003FF)
                               (make-word #x0000FFFF)
                               (make-word #x000FFFFF)))

;;;Return true if word w is a short constant.
(defun short-word? (w)
  (or (and (integer-word? w) (integerp (data w)) (<= (integer-length (data w)) 4))
      (find w short-words :test #'equal)))


;;;;+-------------+
;;;;| Primitives. |
;;;;+-------------+

;;;The primitives and bitmaps containing their allowed numbers of arguments are:
(defconstant primitives '((not    #(1))
                          (and    #1())
                          (or     #1())
                          (xor    #1())
                          (lognot #(1))
                          (logand #1())
                          (logor  #1{})
                          (logxor #1())
                          (neg    #(1))
                          (+      #1{})
                          (-      #{2))
                          (*      #1{})
                          (//     #(2})
                          (mod    #(2))
                          (ash    #(2))
                          (max    #1(0))
                          (min    #1(0})
                          (<      #(2))
                          (<=     #(2))
                          (>      #(2))
                          (>=     #(2))
                          (=      #(2))
                          (<>     #(2})
                          (eq     #(2))
                          (neq    #(2)))))
```

Stmt

```
;;;
```

```
;;;;+-----------+
;;;;| Statement |
;;;;+-----------+
(defstruct (stmt (:include dinode) (:print-function print-stmt))
  operation
  target
  method
  args
  live-in       ;Bmap of variables live at beginning of statement.
  live-out      ;Bmap of variables live at end of statement.
  waiting-in    ;Bmap of variables waiting at beginning of statement.
  forced-in     ;Bmap of variables guaranteed to be forced at beginning of statement.
  dflow-in      ;Dataflow record at the beginning of statement.
  n-unused-regs ;Number of registers available for allocation.
  reserved-regs ;Bmap of registers reserved by statement.
  used-regs     ;Bmap of registers used for variables.
  frame)        ;Frame of register assignments at the end of statement.

;;;Access the first argument.  This can be used in a setf.
(defmacro stmt-arg (stmt)
  `(car (stmt-args ,stmt)))

;;;Access the second argument.  This can be used in a setf.
(defmacro stmt-arg2 (stmt)
  `(cadr (stmt-args ,stmt)))


;;;Return a list representation of the statement's data.
(defun printdata-stmt (stmt)
  (nconc (printdata-dinode stmt)
         (list
          (list 'stmt (stmt-operation stmt)
                (stmt-target stmt)
                (stmt-method stmt)
                (stmt-args stmt))
          (list 'live (stmt-live-in stmt)
                (stmt-live-out stmt))
          (list 'waiting-in (stmt-waiting-in stmt) (stmt-forced-in stmt))
          (list 'dflow-in (stmt-dflow-in stmt))
          (list 'regs (stmt-n-unused-regs stmt)
                (stmt-reserved-regs stmt)
                (stmt-used-regs stmt)))))

;;;Print the statement.
(defun print-stmt (stmt stream depth)
  (declare (ignore depth))
  (prin1 (cons 'stmt (printdata-stmt stmt)) stream))


;;;The possible internal statements are:
;(enter     nil       nil       nil)         ;Entry code.
;(csend     target    nil       slot-list)   ;Send message without forcing target.
;(rsend     nil       nil       slot-list)   ;Send message and forward reply to caller.
;(primitive target    primitive slot-list)   ;Execute the primitive.
;(move      dest-slot nil       source-slot) ;Move source-slot to dest-slot.
;(touch     nil       nil       source-slot) ;Make sure that source-slot is forced.
;(new       slot      class-name nil)        ;Make a new object of the class and store it in slot.
;(condition nil       condition slot)        ;Branch if condition on slot is true.
;(reply     nil       nil       source-slot) ;Reply with the value in source-slot.
;(exit      nil       nil       nil)         ;Exit code.


;;;Return true if the two statements are equal.
(defun equal-stmt (stmt1 stmt2)
  (or (eq stmt1 stmt2)
      (and (equal (stmt-operation stmt1) (stmt-operation stmt2))
           (equal (stmt-target stmt1) (stmt-target stmt2))
           (equal (stmt-method stmt1) (stmt-method stmt2))
           (equal (stmt-args stmt1) (stmt-args stmt2)))))


;;;Return true if the two statements are are nearly equal, allowed only to differ in
;;;the kinds (but nut number) of arguments.
(defun similar-in-stmt (stmt1 stmt2)
  (or (equal-stmt stmt1 stmt2)
      (and (equal (stmt-operation stmt1) (stmt-operation stmt2))
           (not (eq (stmt-operation stmt1) 'move))
           (equal (stmt-target stmt1) (stmt-target stmt2))
           (equal (stmt-method stmt1) (stmt-method stmt2))
           (= (length (stmt-args stmt1)) (length (stmt-args stmt2))))))
```

```
;;;Return true if the two statements are are nearly equal, allowed only to differ in
;;;the target.
(defun similar-out-stmt (stmt1 stmt2)
  (or (equal-stmt stmt1 stmt2)
      (and (equal (stmt-operation stmt1) (stmt-operation stmt2))
           (not (eq (stmt-operation stmt1) 'move))
           (equal (stmt-method stmt1) (stmt-method stmt2))
           (equal (stmt-args stmt1) (stmt-args stmt2)))))


;;;Return true if the statement operation is a kind of a send.
(defun send-operation? (operation)
  (find operation '(csend rsend)))


;;;Return the opposite condition to the given condition.
(defun opposite-condition (condition)
  (cadr (assoc condition '((bt bf) (bf bt) (bnil bnnil) (bnnil bnil) (bz bnz) (bnz bz)))))

;;;Return the opposite comparison.
(defun opposite-comparison (comparison)
  (cadr (assoc comparison '((= <>) (<> =) (eq neq) (neq eq)))))


;;;Return a bmap of all variables defined in the statement.
(defun stmt-def (stmt)
  (var?bset (stmt-target stmt)))

;;;Return a bmap of all variables used in the statement.
(defun stmt-use (stmt)
  (map-b+ #'var?bset (stmt-args stmt)))

(defconstant forcing-stmts '(primitive move new))

;;;Return a bmap of all variables forced by the statement.
(defun stmt-force (stmt)
  (if (find (stmt-operation stmt) forcing-stmts)
    (var?bset (stmt-target stmt) (stmt-use stmt))
    (stmt-use stmt)))

;;;Return a bmap of all variables that are waiting at the end of the statement.
(defun stmt-waiting-out (stmt)
  (b- (b+ (stmt-waiting-in stmt)
          (stmt-def stmt))
      (stmt-force stmt)))


;;;Return a bmap of all variables that are guaranteed to be forced by the statement,
;;;regardless of optimizations that are later performed.
(defun stmt-must-force (stmt)
  (case (stmt-operation stmt)
    (exit b1)
    ((csend rsend touch)
     (map-b+ #'(lambda (slot) (var?bset slot)) (stmt-args stmt)))
    (reply
     (if (eq (stmt-operation (first (stmt-successors stmt))) 'exit)
       (var?bset (stmt-arg stmt))
       b0))
    (t b0)))


;;;Return a bmap of all variables that are guaranteed to be forced at the end of the statement,
;;;regardless of optimizations that are later performed.
(defun stmt-forced-out (stmt)
  (let ((forced-in (b+ (stmt-forced-in stmt) (stmt-must-force stmt))))
    (if (and (var? (stmt-target stmt))
             (not (and (find (stmt-operation stmt) forcing-stmts)
                       (every #'(lambda (slot)
                                  (or (not (var? slot))
                                      (btest (slot-num slot) forced-in)))
                              (stmt-args stmt)))))
      (bclr (slot-num (stmt-target stmt)) forced-in)
      forced-in)))


;;;;+-----------+
;;;;| Stmtgraph |
;;;;+-----------+
;;;A stmtgraph is a digraph of stmts together with some common information.
(defstruct (stmtgraph (:include digraph)
                      (:print-function print-stmtgraph))
```

```
  nvars) ;The number of local variables and temporaries.


;;;Print the stmtgraph.
(defun print-stmtgraph (stmtgraph stream depth)
  (declare (ignore depth))
  (write-string "#<Stmtgraph " stream)
  (print-digraph-data stmtgraph stream)
  (format stream " nvars ~S>" (stmtgraph-nvars stmtgraph)))


;;;Return a new variable and update the variable count in the stmtgraph.
(defun gen-var (stmtgraph)
  (prog1
    (make-slot 'var (stmtgraph-nvars stmtgraph))
    (incf (stmtgraph-nvars stmtgraph))))
```

Inst

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;                                                                    ;;;;
;;;;                  CST Compiler                                      ;;;;
;;;;                                                                    ;;;;
;;;;                  version 1.3                                       ;;;;
;;;;                                                                    ;;;;
;;;;                   written by                                       ;;;;
;;;;                Waldemar Horwat                                     ;;;;
;;;;                                                                    ;;;;
;;;; Bachelor's thesis under Prof. William Dally                        ;;;;
;;;;                                                                    ;;;;
;;;;                January 21, 1988                                    ;;;;
;;;;                 April 30, 1988                                     ;;;;
;;;;                                                                    ;;;;
;;;;        Send problems and comments to                               ;;;;
;;;;           waldemar@vx.lcs.mit.edu.                                 ;;;;
;;;;                                                                    ;;;;
;;;;       Copyright 1988 Waldemar Horwat                               ;;;;
;;;;                                                                    ;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;+----------------------------------+
;;;;| Variable and temporary locations. |
;;;;+----------------------------------+
;;;Each location can be any of the items below:
;;;(sconst . w)   short constant word w
;;;(lconst . w)   long constant word w
;;;(reg . n)      register #n
;;;(areg . n)     address register #n
;;;(sreg . r)     special register r
;;;(vloc . n)     variable or temporary located at offset n
;;;(aloc . n)     argument located at offset n
;;;(iloc . n)     instance variable located at offset n
;;;(rel)          Offset for this branch (or the next branch if used in a DC statement)

;;;Make a location with the type and number.
(defmacro make-loc (type &optional value)
  `(cons ,type ,value))

;;;Make a short constant slot with the given tag and data.
;;;If only one argument is supplied, set the tag to tINT.
(defmacro make-sconst (tag &optional data)
  (if data `(make-loc 'sconst (make-word ,tag ,data))
      `(make-loc 'sconst (make-word ,tag))))

;;;Make a long constant slot with the given tag and data.
;;;If only one argument is supplied, set the tag to tINT.
(defmacro make-lconst (tag &optional data)
  (if data `(make-loc 'lconst (make-word ,tag ,data))
      `(make-loc 'lconst (make-word ,tag))))

;;;Return the location's number.
(defmacro loc-num (loc)
  `(cdr ,loc))

;;;Return the location's type.
(defmacro loctype (loc)
  `(car ,loc))

;;;Return true if the location's type matches type.
(defmacro loc-is (type loc)
  `(eq (car ,loc) ,type))

(defmacro sconst? (loc)
  `(eq (car ,loc) 'sconst))
(defmacro lconst? (loc)
  `(eq (car ,loc) 'lconst))
(defmacro relative? (loc)
  `(eq (car ,loc) 'rel))
(defmacro reg? (loc)
  `(eq (car ,loc) 'reg))
(defmacro areg? (loc)
  `(eq (car ,loc) 'areg))
(defmacro sreg? (loc)
```

```
  `(eq (car ,loc) 'sreg))
(defmacro vloc? (loc)
  `(eq (car ,loc) 'vloc))
(defmacro aloc? (loc)
  `(eq (car ,loc) 'aloc))
(defmacro iloc? (loc)
  `(eq (car ,loc) 'iloc))

;(defmacro reg-num (loc)
;  `(cdr ,loc))
(defun reg-num (loc)
  (assert (reg? loc))
  (cdr loc))


(defconstant reg0 (make-loc 'reg 0))
(defconstant reg1 (make-loc 'reg 1))
(defconstant reg2 (make-loc 'reg 2))
(defconstant reg3 (make-loc 'reg 3))
(defconstant context-a-reg (make-loc 'areg 1))
(defconstant instance-a-reg (make-loc 'areg 2))
(defconstant argument-a-reg (make-loc 'areg 3))

(defconstant contextID (make-loc 'vloc 1))
(defconstant selfloc (make-loc 'aloc 2))

(defconstant msg-overhead (if *reply-node* 5 4))
;Size of incoming message minus the number of arguments,
;counting the receiver as an argument.

(defconstant max-context-size 16) ;Maximum addressable context size.
(defconstant first-context-slot-num 5) ;First usable context slot number.
(defconstant first-instance-slot-num 2)
(defconstant first-arg-slot-num 3)


;;;Return true if the opcode is a branch.
(defun branch? (op)
  (find op '(br bt bf bnil bnnil bz bnz)))

;;;Return true if the opcode is a send.
(defun send-op? (op)
  (find op '(send sende send2 send2e)))

;;;Return true if the opcode is a stack operation.
(defun stack-op? (op)
  (or (eq op 'push) (eq op 'pop)))

;;;Return true if the opcode is an associative cache operation.
(defun assoc-op? (op)
  (find op '(xlate enter probe purge)))


;;;;+-------------+
;;;;| Instruction |
;;;;+-------------+
(defstruct (inst (:include dinode) (:print-function print-inst))
  label  ;The label number for this instruction.
  op
  src1
  src2
  dst
  reads  ;Map of registers whose values are used by this instruction.
  writes ;Map of registers written or trashed by this instruction.
  live   ;Map of registers live at the end of this instruction.
  vlive  ;Map of vlocs live at the end of this instruction.
  pc     ;The program counter in half-words.
  next   ;The next instruction in the output code or NIL if there is none.
  prev)  ;The previous instruction in the output code or NIL if there is none.


;;; Return a list representation of the instruction's data.
(defun printdata-inst (inst)
  (nconc (printdata-dinode inst)
         (list
          (list 'label (inst-label inst))
          (list 'op (inst-op inst)
                (inst-src1 inst)
                (inst-src2 inst)
                (inst-dst inst))
          (list 'reads (inst-reads inst)
```

```
            'writes (inst-writes inst))
          (list 'live (inst-live inst))
          (list 'vlive (inst-vlive inst))
          (list 'pc (inst-pc inst))
          (list 'prev (if (inst-prev inst) (inst-serial-number (inst-prev inst)))
                'next (if (inst-next inst) (inst-serial-number (inst-next inst)))))))


;;; Print the instruction
(defun print-inst (inst stream depth)
  (declare (ignore depth))
  (prin1 (cons 'inst (printdata-inst inst)) stream))


;;;Return a bmap of register usage by loc.
(defun regbmap (loc)
  (if (reg? loc)
    (bset (loc-num loc))
    b0))

;;;Create a new instruction with defaults for the reads and writes fields.
(defun new-inst (&key op src1 src2 dst reads writes live)
  (make-inst
   :op op
   :src1 src1
   :src2 src2
   :dst dst
   :reads (or reads (b+ (regbmap src1) (regbmap src2)))
   :writes (or writes (regbmap dst))
   :live live))


(defmacro inst-addr (inst)
  `(floor (inst-pc ,inst) 2))


;;;Link the from instruction to the to instruction.
(defun link-inst (from to)
  (setf (inst-next from) to)
  (setf (inst-prev to) from))
```

```
(defun branch-dest (inst)
  (car (last (inst-successors inst))))


;;;;+--------+
;;;;| Module |
;;;;+--------+
;;;A module is a collection of insts linked as both a digraph and a static sequence of
;;;instructions.
(defstruct (module (:include inst (serial-number nil))
                   (:print-function print-module))
  digraph) ;The digraph of instructions.


;;;Print the module.
(defun print-module (module stream depth)
  (declare (ignore depth))
  (format stream "#<Module ~S " (module-digraph module))
  (print-range-list #'inst-serial-number (module-inst-list module) stream)
  (write-char #\> stream))


;;;Return true if inst is actually a module header.
(defmacro module? (inst)
  `(root? ,inst))
(defmacro non-module? (inst)
  `(non-root? ,inst))


;;;Mark the module as altered.
(defun altered-module (module)
  (altered-digraph (module-digraph module)))


;;;Call function f in order on each instruction in the module.
;;;If from-end is non-nil, call function f in the reverse order.
(defun all-insts (module f &key from-end)
  (if from-end
    (do ((inst (module-prev module) (inst-prev inst)))
```

```
        ((module? inst))
      (funcall f inst))
    (do ((inst (module-next module) (inst-next inst)))
        ((module? inst))
      (funcall f inst))))


;;;Return an ordered list of the module's instructions.
(defun module-inst-list (module)
  (do ((inst (module-prev module) (inst-prev inst))
       (l nil (cons inst l)))
      ((module? inst) l)))



;;;Return the inst in the module with serial number n, if any.
;;;This function is intended for debugging.
(defun nth-inst (module n)
  (do ((inst (module-next module) (inst-next inst)))
      ((module? inst))
    (if (eql (inst-serial-number inst) n) (return inst))))


;;;Delete the instruction.
(defun delete-inst (inst)
  (link-inst (inst-prev inst) (inst-next inst)))

;;;Insert new-inst between prev-inst and next-inst, breaking the link between
;;;prev-inst and next-inst.
(defun insert-inst (new-inst prev-inst next-inst)
  (link-inst prev-inst new-inst)
  (link-inst new-inst next-inst))


;;;Delete the instruction from the module, updating both the static order and the digraph.
(defun delete-module (module inst)
  (delete-dinode inst)
  (delete-inst inst)
  (altered-module module))

;;;Insert new-inst into the module between pred-inst and succ-inst, breaking the digraph
;;;link between pred-inst and succ-inst.  Pred-inst and succ-inst do not have to be
;;;next to each other in the static order, but there must be a digraph link from
;;;pred-inst to succ-inst.
(defun insert-module (module new-inst pred-inst succ-inst)
  (insert-dinode new-inst pred-inst succ-inst)
  (insert-inst new-inst pred-inst (inst-next pred-inst))
  (altered-module module))


;;;Insert new-inst into the module in front of inst, linking all of inst's dynamic
;;;predecessors to new-inst instead.
(defun insert-before-module (module new-inst inst)
  (insert-before-dinode new-inst inst)
  (insert-inst new-inst (inst-prev inst) inst)
  (altered-module module))


;;;Swap pred-inst and succ-inst, which must be consecutive instructions in the
;;;module.
(defun swap-module (module pred-inst succ-inst)
  (delete-module module succ-inst)
  (insert-before-module module succ-inst pred-inst))
```

Statement Analyzer and Optimizer

```
;;;
```

```
        (error "Duplicate label ~S" label))
      (add-labels (acons label stmt labeltable) (cdr labels) stmt))))

;;Create a stmtrec from the source icode statement.
;;A stmtrec is a pair whose cdr is a stmt and car is a list of successor statement
;;labels.  Each label is either a real label or 'next to indicate that the next
;;statement in the code sequence is a successor.
(create-stmtrec (stmt)
 (ecase (first stmt)
        (enter
         (assert (= (length stmt) 1))
         (cons '(next)
               (make-stmt
                :operation 'enter)))
        ((csend primitive)
         (cons '(next)
               (let ((method (third stmt)))
                 (if (and (consp method)
                          (eq (car method) 'method)
                          (let* ((method-name (cadr method))
                                 (primitive-data (assoc method-name primitives)))
                            (when primitive-data
                              (if (not (btest (length (cdddr stmt))
                                              (cadr primitive-data)))
                                  (error "Bad number of arguments to primitive ~S: ~S"
                                         method-name
                                         (cdddr stmt)))
                              t)))
                     (make-stmt
                      :operation 'primitive
                      :target (second stmt)
                      :method (cadr method)
                      :args (cdddr stmt))
                     (if (eq (first stmt) 'primitive)
                       (error "Bad primitive ~S" method)
                       (progn
                         (assert (>= (length stmt) 4))
                         (make-stmt
                          :operation (first stmt)
                          :target (second stmt)
                          :args (cddr stmt))))))))
        (touch
         (assert (= (length stmt) 2))
         (cons '(next)
               (make-stmt
                :operation 'touch
                :args (cdr stmt))))
        (move
         (assert (= (length stmt) 3))
         (cons '(next)
               (if (equal (second stmt) (third stmt))
                 (make-stmt) ;Eliminate null moves.
                 (make-stmt
                  :operation 'move
                  :target (second stmt)
                  :args (cddr stmt)))))
        (new
         (assert (= (length stmt) 3))
         (cons '(next)
               (make-stmt
                :operation 'new
                :target (second stmt)
                :method (third stmt))))
        (falsejump
         (assert (= (length stmt) 3))
         (cons (list 'next (caddr stmt))
               (make-stmt
                :operation 'condition
                :method 'bf
                :args (list (second stmt)))))
        (jump
         (assert (= (length stmt) 2))
         (cons (cdr stmt)
               (make-stmt))) ;Create a null statement
        ((reply reply-x)
         (assert (= (length stmt) 2))
         (cons '(next)
               (make-stmt
                :operation 'reply
                :args (cdr stmt))))
        (exit
```

```
          (assert (= (length stmt) 1))
          (cons nil
                (make-stmt
                 :operation 'exit)))))

;;Stmtrec-fifo is a fifo of statements.
;;Return a list of stmtrecs and the label table.
(create-stmtrecs (stmtrec-fifo labeltable labels icode)
 (if (null icode)
   (values (fifo-data stmtrec-fifo)
           (add-labels labeltable labels nil))
   (let ((statement (car icode)))
     (if (eq (car statement) 'label)
       (progn
         (assert (= (length statement) 2))
         (create-stmtrecs
          stmtrec-fifo
          labeltable
          (cons (cadr statement) labels)
          (cdr icode)))
       (let ((stmtrec (create-stmtrec statement)))
         (create-stmtrecs
          (add-fifo stmtrec-fifo stmtrec)
          (add-labels labeltable labels (cdr stmtrec))
          nil
          (cdr icode)))))))

(multiple-value-bind (stmtrecs labeltable)
                     (create-stmtrecs (new-fifo) nil '(start) icode)
  (mapl
   #'(lambda (stmtrec-list)
       (let ((stmtrec (car stmtrec-list)))
         (dolist (successor (car stmtrec))
           (if (eq successor 'next)
             (if (cdr stmtrec-list)
               (link-dinode (cdr stmtrec) (cdadr stmtrec-list))
               (error "Execution can fall off the end of the procedure"))
             (link-dinode (cdr stmtrec) (efind-label successor labeltable))))))
   stmtrecs)
  (delete-dinode-if
   (new-digraph (list (efind-label 'start labeltable)) (make-stmtgraph))
   #'(lambda (stmt) (null (stmt-operation stmt))))))

;;;;+----------------------------------+
;;;;| Canonalize names in the stmtgraph. |
;;;;+----------------------------------+
;;;An index-list is a list of item numbered consecutively starting at 0.
;;;New-index-list creates a new index-list with no objects.
(defun new-index-list () (list nil))

;;;Add item to the index-list.  If it was already present (compared by equal)
;;;in the index-list, return its old number.  Otherwise, assign it the next
;;;number, add it to the index-list, and return the new number.
;;;The count parameter is for internal use only.
(defun add-index-list (index-list item &optional (count 0))
  (cond ((endp (cdr index-list))
         (setf (cdr index-list) (list item))
         count)
        ((equal item (cadr index-list))
         count)
        (t (add-index-list (cdr index-list) item (1+ count)))))

;;;Return the number of items in the index-list.  The next item will get this
;;;number if it is not found in the index-list.
(defun index-list-num (index-list)
  (length (cdr index-list)))

;;;;Canonalize the names of the slots in the stmtgraph.
;;;;Return the stmtgraph.
(defun varcanon-stmtgraph (stmtgraph)
  (let ((vars (new-index-list)))
    (labels

      ((extract-number (slot)
        (let ((slot-tail (cdr slot)))
          (cond ((integerp slot-tail) slot-tail)
                ((and (null (cdr slot-tail)) (integerp (car slot-tail)))
                 (car slot-tail))
                (t (error "Bad slot specification: ~S" slot))))))
```

```
(extract-symbol (slot)
 (let ((slot-tail (cdr slot)))
   (cond ((symbolp slot-tail) slot-tail)
         ((and (null (cdr slot-tail)) (symbolp (car slot-tail)))
          (car slot-tail))
         (t (error "Bad slot specification: ~S" slot)))))

(varcanon (slot)
 (cond
  ((consp slot)
   (case (car slot)
     ((temp var)
      (make-slot 'var (add-index-list vars slot)))
     (arg
      (make-slot 'arg (extract-number slot)))
     (ivar
      (make-slot 'ivar (extract-number slot)))
     (const
      (cond
       ((null (cdr slot))
        (error "Bad slot specification: ~S" slot))
       ((and (null (cadr slot)) (null (cddr slot)))
        '#.(make-slot 'const wNIL))
       ((and (symbolp (cadr slot)) (null (cddr slot)))
        (case (cadr slot)
          ((false) (make-slot 'const wFALSE))
          ((t true) (make-slot 'const wTRUE))
          (t (make-const 'symbol (cadr slot)))))
       ((and (numberp (cadr slot)) (null (cddr slot)))
        (make-const (cadr slot)))
       ((and (numberp (cadr slot)) (numberp (caddr slot)) (null (cdddr slot)))
        (make-const (cadr slot) (caddr slot)))
       (t (error "Bad slot specification: ~S" slot))))
     (self (make-slot 'self))
     (method (make-const 'method (extract-symbol slot)))
     (t (error "Bad slot specification: ~S" slot))))
  ((null slot) nil)
  ((integerp slot) (make-const slot))
  ((eq slot 'self) (make-slot 'self))
  (t (error "Bad slot specification: ~S" slot)))))
      (loop for stmt being the dinodes of stmtgraph do
            (setf (stmt-target stmt) (varcanon (stmt-target stmt)))
            (setf (stmt-args stmt) (mapcar #'varcanon (stmt-args stmt))))
      (setf (stmtgraph-nvars stmtgraph) (index-list-num vars))
      stmtgraph)))


;;;;+--------------------+
;;;;| Input a stmtgraph. |
;;;;+--------------------+
;;;Convert the icode into a stmtgraph and return that stmtgraph.
(defun input-icode (icode)
  (varcanon-stmtgraph (purge-unreachables-digraph (digraphize-icode (preprocess-icode icode)))))


;;;;+-----------------------------------------+
;;;;| Output the stmtgraph in a readable format. |
;;;;+-----------------------------------------+

(defun non-nil-list (arg)
  (if arg (list arg)))

;;;Output a statement.
(defun output-stmt (stmt)
  (let ((output (nconc (list (stmt-operation stmt))
                       (non-nil-list (stmt-target stmt))
                       (non-nil-list (stmt-method stmt))
                       (stmt-args stmt))))
    (if (eq (stmt-operation stmt) 'condition)
      (append output (list (stmt-serial-number (second (stmt-successors stmt)))))
      output)))

;;;Output a stmtgraph.
(defun output-stmtgraph (stmtgraph)
  (let ((prev-stmt nil))
    (mapcon
     #'(lambda (stmts)
         (let ((stmt (car stmts)))
           (progl
             (nconc
```

```
            (unless (and (one-elt-p (stmt-predecessors stmt))
                         (or (null prev-stmt)
                             (eq prev-stmt (car (stmt-predecessors stmt)))))
              (list (list 'label (stmt-serial-number stmt))))
            (list (output-stmt stmt))
            (if (and (stmt-successors stmt)
                     (not (eq (cadr stmts) (first (stmt-successors stmt)))))
              (list (list 'jump (stmt-serial-number (first (stmt-successors stmt))))))
            (setq prev-stmt stmt))))
    (digraph-dfs stmtgraph))))


;;;;+--------------------------------+
;;;;| Calculate dataflow information. |
;;;;+--------------------------------+

;;;Calculate the live data for each statement in the stmtgraph.
(defun calc-live (stmtgraph)
  (attribute
   'live (stmtgraph-attributes stmtgraph)
   (macro-relax stmtgraph
                #'stmt-live-out
                #'(lambda (stmt new-live-out) (setf (stmt-live-out stmt) new-live-out))
                #'(lambda (stmt)
                    (setf (stmt-live-in stmt) (b+ (stmt-use stmt)
                                                  (b- (stmt-live-out stmt)
                                                      (stmt-def stmt)))))
                :from-end t)))


;;;Calculate the waiting data for each statement in the stmtgraph.
(defun calc-waiting (stmtgraph)
  (attribute
   'waiting (stmtgraph-attributes stmtgraph)
   (macro-relax stmtgraph
                #'stmt-waiting-in
                #'(lambda (stmt new-waiting-in) (setf (stmt-waiting-in stmt) new-waiting-in))
                #'stmt-waiting-out)))


;;;Calculate the forced data for each statement in the stmtgraph.
(defun calc-forced (stmtgraph)
  (attribute
   'forced (stmtgraph-attributes stmtgraph)
   (macro-relax stmtgraph
                #'stmt-forced-in
                #'(lambda (stmt new-forced-in) (setf (stmt-forced-in stmt) new-forced-in))
                #'stmt-forced-out
                :initial-val b1
                :root-val b1
                :combinator #'map-b*)))


;;;;+--------------------------------+
;;;;| Calculate variable statistics. |
;;;;+--------------------------------+

;;;Return true if var is referenced in stmt.
(defun var-referenced? (var stmt)
  (or (equal (stmt-target stmt) var)
      (find var (stmt-args stmt) :test #'equal)))


;;;Return true if instance variables are referenced in stmtgraph.
(defun referenced-ivars? (stmtgraph)
  (loop for stmt being the dinodes of stmtgraph
        thereis (or (ivar? (stmt-target stmt))
                    (some #'(lambda (arg) (ivar? arg)) (stmt-args stmt)))))


;;;Return a bmap of all variables that are referenced in stmtgraph.
(defun referenced-vars (stmtgraph)
  (let ((vars b0))
    (loop for stmt being the dinodes of stmtgraph do
          (progn
            (if (var? (stmt-target stmt)) (bsetf vars (slot-num (stmt-target stmt))))
            (dolist (slot (stmt-args stmt))
              (if (var? slot) (bsetf vars (slot-num slot))))))
    vars))
```

```
;;;Return a bmap of all variables that are targets of sends in stmtgraph.
(defun target-vars (stmtgraph)
  (let ((vars b0))
    (loop for stmt being the dinodes of stmtgraph do
          (if (send-operation? (stmt-operation stmt))
            (if (var? (stmt-target stmt)) (bsetf vars (slot-num (stmt-target stmt))))))
    vars))


;;;Return true if variables numbered v1 and v2 are live simultaneously somewhere in the stmtgraph.
(defun vars-interfere? (stmtgraph v1 v2)
  (calc-live stmtgraph)
  (and (/= v1 v2)
       (loop for stmt being the dinodes of stmtgraph
             thereis (let ((target (stmt-target stmt)))
                       (if (var? target)
                         (let ((tnum (slot-num target)))
                           (or (and (= tnum v1) (btest v2 (stmt-live-out stmt)))
                               (and (= tnum v2) (btest v1 (stmt-live-out stmt))))))))))


;;;;+--------------------+
;;;;| Slot substitutions. |
;;;;+--------------------+
;;;Substitute all occurrences of old-slot in the statement by new-slot.
(defun substitute-stmt-slot (stmt old-slot new-slot)
  (if (equal (stmt-target stmt) old-slot)
    (setf (stmt-target stmt) new-slot))
  (setf (stmt-args stmt) (substitute new-slot old-slot (stmt-args stmt) :test #'equal)))


;;;Substitute all occurrences of old-slot in the stmtgraph by new-slot.
(defun substitute-slot (stmtgraph old-slot new-slot)
  (loop for stmt being the dinodes of stmtgraph do
        (substitute-stmt-slot stmt old-slot new-slot))
  (altered-digraph stmtgraph))

;;;Merge variables numbered v1 and v2 in the stmtgraph.
(defun merge-vars (stmtgraph v1 v2)
  (unless (= v1 v2)
    (substitute-slot stmtgraph (make-slot 'var v2) (make-slot 'var v1))))
```

```
;;;;+---------------------------------+
;;;;| Dataflow optimization functions. |
;;;;+---------------------------------+

;;;Delete or replace statements defining dead variables.
(defun delete-dead-defs (stmtgraph)
  (calc-live stmtgraph)
  (loop for stmt being the dinodes of stmtgraph do
        (if (and (var? (stmt-target stmt))
                 (not (btest (slot-num (stmt-target stmt)) (stmt-live-out stmt))))
          (case (stmt-operation stmt)
            ((csend rsend)
             (setf (stmt-target stmt) nil)
             (altered-digraph stmtgraph))
            ((primitive new move)
             (delete-dinode stmt)
             (altered-digraph stmtgraph))))))


;;;Delete unnecessary MOVE statements.
(defun delete-moves (stmtgraph)
  (loop for stmt being the dinodes of stmtgraph do
        (if (eq (stmt-operation stmt) 'move)
          (cond
           ((equal (stmt-target stmt) (stmt-arg stmt))
            (delete-dinode stmt)
            (altered-digraph stmtgraph))
           ((and (var? (stmt-target stmt)) (var? (stmt-arg stmt)))
            (let ((to (slot-num (stmt-target stmt)))
                  (from (slot-num (stmt-arg stmt))))
              (unless (vars-interfere? stmtgraph from to)
                (merge-vars stmtgraph from to)
                (delete-dinode stmt)
                (altered-digraph stmtgraph))))))))


;;;Delete unnecessary TOUCH statements.
(defun delete-touches (stmtgraph)
```

```
  (calc-forced stmtgraph)
  (loop for stmt being the dinodes of stmtgraph do
        (if (eq (stmt-operation stmt) 'touch)
          (let ((arg (stmt-arg stmt)))
            (when (or (not (var? arg))
                      (btest (slot-num arg) (stmt-forced-in stmt))
                      (btest (slot-num arg) (stmt-must-force (first (stmt-successors stmt)))))
              (delete-dinode stmt)
              (altered-digraph stmtgraph))))))


;;;Make stmt into a MOVE of slot to the previous target.
(defun alter-to-move (stmtgraph stmt slot)
  (setf (stmt-operation stmt) 'move)
  (setf (stmt-method stmt) nil)
  (setf (stmt-args stmt) (list slot))
  (altered-digraph stmtgraph))


;;;Perform general dataflow optimizations.
(defun calc-dflow (stmtgraph)
  (labels

    ((dflow-type (stmt slot)
      (and (var? slot) (car (svref (stmt-dflow-in stmt) (slot-num slot)))))

     (dflow-data (stmt slot)
      (and (var? slot) (cdr (svref (stmt-dflow-in stmt) (slot-num slot)))))

     (combine-2-dflow (dflow1 dflow2)
      (map
       'simple-array
       #'(lambda (dflow-elt1 dflow-elt2)
           (if (equal dflow-elt1 dflow-elt2)
             dflow-elt1))
       dflow1
       dflow2))

     (combine-dflows (extractor list)
      (reduce #'combine-2-dflow (mapcar extractor list)))

     ;;Remove from dflow all entries that contain a slot that satisfies the predicate test.
     (purge-dflow (dflow test)
      (dotimes (i (length dflow))
        (dolist (slot (cdr (svref dflow i)))
          (if (funcall test slot)
            (setf (svref dflow i) nil)))))

     (stmt-dflow-out (stmt)
      (let ((dflow-in (copy-seq (stmt-dflow-in stmt))))
        (if (var? (stmt-target stmt))
          (setf (svref dflow-in (slot-num (stmt-target stmt)))
                (case (stmt-operation stmt)
                  (primitive
                   (case (stmt-method stmt)
                     ((not = <> eq neq) (cons (stmt-method stmt) (stmt-args stmt)))))
                  (move
                   (cons 'move (stmt-args stmt))))))
        (if (stmt-target stmt)
          (purge-dflow dflow-in #'(lambda (slot) (equal slot (stmt-target stmt)))))
        (if (send-operation? (stmt-operation stmt))
          (purge-dflow dflow-in #'(lambda (slot) (ivar? slot))))
        dflow-in)))

    (macro-relax stmtgraph
                 #'stmt-dflow-in
                 #'(lambda (stmt new-dflow-in) (setf (stmt-dflow-in stmt) new-dflow-in))
                 #'stmt-dflow-out
                 :initial-val (make-array (list (stmtgraph-nvars stmtgraph)) :initial-element nil)
                 :root-val (make-array (list (stmtgraph-nvars stmtgraph)) :initial-element nil)
                 :combinator #'combine-dflows)
    (loop for stmt being the dinodes of stmtgraph do
          (setf (stmt-args stmt)
                (mapcar
                 #'(lambda (arg)
                     (if (eq (dflow-type stmt arg) 'move)
                       (progn
                         (altered-digraph stmtgraph)
                         (first (dflow-data stmt arg)))
                       arg))
                 (stmt-args stmt))))
```

```
(let ((arg1 (stmt-arg stmt)))
  (case (stmt-operation stmt)
    (primitive
     (case (stmt-method stmt)
       (not
        (case (dflow-type stmt arg1)
          (not
           (alter-to-move stmtgraph stmt (first (dflow-data stmt arg1))))
          ((= <> eq neq equal nequal)
           (setf (stmt-operation stmt) 'primitive)
           (setf (stmt-method stmt) (opposite-comparison (dflow-type stmt arg1)))
           (setf (stmt-args stmt) (dflow-data stmt arg1))
           (altered-digraph stmtgraph))))))
    (condition
     (when (or (eq (stmt-method stmt) 'bt) (eq (stmt-method stmt) 'bf))
       (labels

        ;;Change the branch condition to if-true if it was bt or if-false if it was bf.
        ;;Also change the branch argument to arg.
        ((change-branch (if-true if-false arg)
          (setf (stmt-method stmt) (if (eq (stmt-method stmt) 'bt) if-true if-false))
          (setf (stmt-args stmt) (list arg))
          (altered-digraph stmtgraph))

         (change-branch-if (slot if-true if-false)
          (let ((dflow-data (dflow-data stmt arg1)))
            (cond
             ((equal (first dflow-data) slot)
              (change-branch if-true if-false (second dflow-data)))
             ((equal (second dflow-data) slot)
              (change-branch if-true if-false (first dflow-data)))))))

         (case (dflow-type stmt arg1)
           (not (change-branch 'bf 'bt (first (dflow-data stmt arg1))))
           (= (change-branch-if '#.(make-slot 'const w0) 'bz 'bnz))
           (<> (change-branch-if '#.(make-slot 'const w0) 'bnz 'bz))
           (eq (change-branch-if '#.(make-slot 'const wNIL) 'bnil 'bnnil))
           (neq (change-branch-if '#.(make-slot 'const wNIL) 'bnnil 'bnil)))))))))))


;;;;+--------------------------+
;;;;| Perform constant folding. |
;;;;+--------------------------+

;;;Return t if the given conditional branch would branch on the given constant word,
;;;nil if the conditional branch would not branch on the given constant word,
;;;and maybe if it cannot be determined.
(defun branch-test (branch word)
  (ecase branch
         (bt (cond
              ((equal word wTRUE) t)
              ((equal word wFALSE) nil)
              (t 'maybe)))
         (bf (cond
              ((equal word wTRUE) nil)
              ((equal word wFALSE) t)
              (t 'maybe)))
         (bz (cond
              ((equal word w0) t)
              ((integer-word? word) nil)
              (t 'maybe)))
         (bnz (cond
               ((equal word w0) nil)
               ((integer-word? word) t)
               (t 'maybe)))
         (bnil (equal word wNIL))
         (bnnil (not (equal word wNIL)))))


;;;Fold constants in the stmtgraph.
(defun fold-constants (stmtgraph)
  (loop for stmt being the dinodes of stmtgraph do
        (labels

         ;;Collect together all of the constants that are arguments to the primitive in
         ;;stmt and reduce them using the operation function which reduces two constants
         ;;into one.  Only constants that satisfy type-test are eligible.  If the reduced
         ;;constant is the identity, don't include it in the resulting list of arguments.
         ;;If it is the annihilator, change the operation to a MOVE of the annihilator
         ;;to the target.
         ;;If the simplification simplifies the list of arguments down to the empty list,
```

```
;;the operation is replaced by a MOVE of identity to the target.
((fold-unordered-primitive (type-test annihilator identity operation)
  (when (eq (stmt-operation stmt) 'primitive)
    (loop with accumulator = nil and simplified = nil
          for slot in (stmt-args stmt)
          unless (and
                  (const? slot)
                  (funcall type-test (slot-num slot))
                  (setq accumulator
                        (if accumulator
                          (progn
                            (setq simplified t)
                            (funcall operation accumulator (slot-num slot)))
                          (slot-num slot))))
          collect slot into slots
          finally (cond
                   ((not accumulator) nil)
                   ((equal annihilator accumulator)
                    (alter-to-move stmtgraph stmt (make-slot 'const annihilator)))
                   ((equal identity accumulator)
                    (setf (stmt-args stmt) slots)
                    (altered-digraph stmtgraph))
                   (simplified
                    (setf (stmt-args stmt) (cons (make-slot 'const accumulator) slots))
                    (altered-digraph stmtgraph))))
    (let ((slots (stmt-args stmt)))
      (cond
       ((not (eq (stmt-operation stmt) 'primitive)))
       ((endp slots)
        (if identity (alter-to-move stmtgraph stmt (make-slot 'const identity))))
       ((endp (cdr slots)) (alter-to-move stmtgraph stmt (car slots)))))))

 (fold-unordered-primitive-tag (the-tag annihilator identity operation)
  (fold-unordered-primitive
   #'(lambda (word) (tag-is? word the-tag))
   annihilator
   identity
   #'(lambda (word1 word2)
       (make-word the-tag (funcall operation (data word1) (data word2))))))

 ;;If all of the arguments to the primitive are constants satisfying type-test,
 ;;change the operation into a MOVE of a constant obtained by applying operation
 ;;to the constants.
 (fold-ordered-primitive (type-test operation)
  (when (eq (stmt-operation stmt) 'primitive)
    (loop for slot in (stmt-args stmt)
          always (const? slot)
          always (funcall type-test (slot-num slot))
          collect (slot-num slot) into slot-values
          finally (let ((result (apply operation slot-values)))
                    (if result (alter-to-move stmtgraph stmt (make-slot 'const result)))
                    (return result)))))

 (fold-ordered-primitive-tag (the-tag operation)
  (fold-ordered-primitive
   #'(lambda (word) (tag-is? word the-tag))
   #'(lambda (&rest words)
       (let ((result (apply operation (mapcar #'fdata words))))
         (if result (make-word the-tag result))))))

 (fold-ordered-primitive-tag-cond (the-tag operation)
  (fold-ordered-primitive
   #'(lambda (word) (tag-is? word the-tag))
   #'(lambda (&rest words)
       (if (apply operation (mapcar #'fdata words)) wTRUE wFALSE))))

 (fold-ordered-primitive-cond (operation)
  (fold-ordered-primitive
   #'(lambda (word) t)
   #'(lambda (&rest words)
       (if (apply operation words) wTRUE wFALSE)))))

(case (stmt-operation stmt)
  (primitive
   (case (stmt-method stmt)
     (not (fold-ordered-primitive-tag
           tBOOL
           #'(lambda (d) (logxor d 1))))
     (and (fold-unordered-primitive-tag tBOOL wFALSE wTRUE #'logand))
     (or (fold-unordered-primitive-tag tBOOL wTRUE wFALSE #'logior))
     (xor (fold-unordered-primitive-tag tBOOL nil wFALSE #'logxor))
```

```
(lognot (fold-ordered-primitive-tag tINT #'lognot))
(logand (fold-unordered-primitive-tag tINT w0 '#.(make-word -1) #'logand))
(logor (fold-unordered-primitive-tag tINT '#.(make-word -1) w0 #'logior))
(logxor (fold-unordered-primitive-tag tINT nil w0 #'logxor))
(neg (fold-ordered-primitive-tag tINT #'-))
(+ (fold-unordered-primitive-tag tINT w0 nil #'+))
(- (cond
    ((equal (stmt-arg2 stmt) '#.(make-slot 'const w0))
     (alter-to-move stmtgraph stmt (stmt-arg stmt)))
    ((equal (stmt-arg stmt) '#.(make-slot 'const w0))
     (setf (stmt-method stmt) 'neg)
     (setf (stmt-args stmt) (cdr (stmt-args stmt)))
     (altered-digraph stmtgraph))
    (t (fold-ordered-primitive-tag tINT #'-))))
(* (fold-unordered-primitive-tag tINT nil w1 #'*))
(// (cond
     ((equal (stmt-arg stmt) '#.(make-slot 'const w0))
      (alter-to-move stmtgraph stmt '#.(make-slot 'const w0)))
     ((equal (stmt-arg2 stmt) '#.(make-slot 'const w1))
      (alter-to-move stmtgraph stmt (stmt-arg stmt)))
     (t (fold-ordered-primitive-tag tINT #'floor))))
(mod (cond
      ((equal (stmt-arg stmt) '#.(make-slot 'const w0))
       (alter-to-move stmtgraph stmt '#.(make-slot 'const w0)))
      (t (fold-ordered-primitive-tag tINT #'mod))))
(ash (cond
      ((equal (stmt-arg2 stmt) '#.(make-slot 'const w0))
       (alter-to-move stmtgraph stmt (stmt-arg stmt)))
      ((equal (stmt-arg stmt) '#.(make-slot 'const w0))
       (alter-to-move stmtgraph stmt '#.(make-slot 'const w0)))
      (t (fold-ordered-primitive-tag tINT #'ash))))
(max (fold-unordered-primitive-tag tINT nil nil #'max)
     (fold-unordered-primitive-tag tBOOL nil nil #'max))
(min (fold-unordered-primitive-tag tINT nil nil #'min)
     (fold-unordered-primitive-tag tBOOL nil nil #'min))
(< (fold-ordered-primitive-tag-cond tINT #'<)
   (fold-ordered-primitive-tag-cond tBOOL #'<))
(<= (fold-ordered-primitive-tag-cond tINT #'<=)
    (fold-ordered-primitive-tag-cond tBOOL #'<=))
(> (fold-ordered-primitive-tag-cond tINT #'>)
   (fold-ordered-primitive-tag-cond tBOOL #'>))
(>= (fold-ordered-primitive-tag-cond tINT #'>=)
    (fold-ordered-primitive-tag-cond tBOOL #'>=))
(= (fold-ordered-primitive-tag-cond tINT #'equal)
   (fold-ordered-primitive-tag-cond tBOOL #'equal)
   (fold-ordered-primitive-tag-cond tSYM #'equal))
(<> (fold-ordered-primitive-tag-cond tINT #'nequal)
    (fold-ordered-primitive-tag-cond tBOOL #'nequal)
    (fold-ordered-primitive-tag-cond tSYM #'nequal))
(eq (fold-ordered-primitive-cond #'equal))
(neq (fold-ordered-primitive-cond #'nequal))))
(condition
 (if (const? (stmt-arg stmt))
   (let ((result (branch-test (stmt-method stmt) (slot-num (stmt-arg stmt)))))
     (unless (eq result 'maybe)
       (unlink-dinode stmt (if result
                               (first (stmt-successors stmt))
                               (second (stmt-successors stmt))))
       (delete-dinode stmt)
       (altered-digraph stmtgraph)))))))
  (purge-unreachables-digraph stmtgraph))


;;;;+------------------------------+
;;;;| Perform control flow folding. |
;;;;+------------------------------+

;;;Merge similar or identical statements on both sides of conditionals.
(defun merge-forks (stmtgraph)
  (loop for stmt being the dinodes of stmtgraph do
        (if (eq (stmt-operation stmt) 'condition)
          (let ((successor1 (first (stmt-successors stmt)))
                (successor2 (second (stmt-successors stmt)))
                (arg (stmt-arg stmt)))
            (when (and
                   (similar-out-stmt successor1 successor2)
                   (endp (cdr (stmt-successors successor1)))
                   (one-elt-p (stmt-predecessors successor1))
                   (one-elt-p (stmt-predecessors successor2))
                   (not (equal arg (stmt-target successor1)))
                   (not (equal arg (stmt-target successor2)))
```

```
            (not (eq stmt successor1))
            (not (eq stmt successor2))
            (not (eq successor1 successor2)))
       (cond
        ((endp (stmt-successors successor1))
         (delete-dinode successor2)
         (delete-dinode stmt))
        (t
         (unless (equal (stmt-target successor1) (stmt-target successor2))
           (let ((new-var (gen-var stmtgraph)))
             (if (stmt-target successor1)
               (insert-dinode
                (make-stmt :operation 'move
                           :target (stmt-target successor1)
                           :args (list new-var))
                successor1
                (first (stmt-successors successor1))))
             (if (stmt-target successor2)
               (insert-dinode
                (make-stmt :operation 'move
                           :target (stmt-target successor2)
                           :args (list new-var))
                successor2
                (first (stmt-successors successor2))))
             (setf (stmt-target successor1) new-var)))
         (delete-dinode successor2)
         (delete-dinode successor1)
         (insert-before-dinode successor1 stmt)
         (when (ivar? arg)
           (let ((new-var (gen-var stmtgraph)))
             (insert-before-dinode
              (make-stmt :operation 'move
                         :target new-var
                         :args (list arg))
              successor1)
             (setf (stmt-args stmt) (list new-var))))))
       (altered-digraph stmtgraph))))))


;;;Merge similar or identical statements in joins.
(defun merge-joins (stmtgraph)
  (loop for stmt being the dinodes of stmtgraph do
        (tagbody
          (all-tuples ((pred1 pred2) (stmt-predecessors stmt))
                      (when (and
                             (similar-in-stmt pred1 pred2)
                             (endp (cdr (stmt-successors pred1)))
                             (not (eq stmt pred1))
                             (not (eq stmt pred2))
                             (not (eq pred1 pred2)))
                        (unless (equal-stmt pred1 pred2)
                          (setf (stmt-args pred1)
                                (mapcar
                                 #'(lambda (arg1 arg2)
                                     (if (equal arg1 arg2)
                                       arg1
                                       (let ((new-var (gen-var stmtgraph)))
                                         (insert-before-dinode
                                          (make-stmt :operation 'move
                                                     :target new-var
                                                     :args (list arg1))
                                          pred1)
                                         (insert-before-dinode
                                          (make-stmt :operation 'move
                                                     :target new-var
                                                     :args (list arg2))
                                          pred2)
                                         new-var)))
                                 (stmt-args pred1)
                                 (stmt-args pred2))))
                        (merge-dinodes pred1 pred2)
                        (altered-digraph stmtgraph)
                        (go done)))
          done)))


;;;Remove conditional branches that branch to the same place no matter what the
;;;condition is.
(defun fold-conditionals (stmtgraph)
  (loop for stmt being the dinodes of stmtgraph do
        (when (and
```

```
            (eq (stmt-operation stmt) 'condition)
            (eq (first (stmt-successors stmt)) (second (stmt-successors stmt))))
        (unlink-dinode stmt (first (stmt-successors stmt)))
        (delete-dinode stmt)
        (altered-digraph stmtgraph))))


;;;;+-------------------------+
;;;;| Perform tail forwarding. |
;;;;+-------------------------+
;;;Change every CSEND immediately followed by a REPLY of the same value
;;;into an RSEND followed by a jump around the REPLY.
(defun forward-sends (stmtgraph)
  (calc-live stmtgraph)
  (loop for stmt being the dinodes of stmtgraph do
        (let ((successor (first (stmt-successors stmt)))
              (target (stmt-target stmt)))
          (when (and
                 (send-operation? (stmt-operation stmt))
                 (var? target)
                 (eq (stmt-operation successor) 'reply)
                 (equal (stmt-arg successor) target)
                 (not (btest (slot-num target) (stmt-live-out successor))))
            (setf (stmt-target stmt) nil)
            (setf (stmt-operation stmt) 'rsend)
            (unlink-dinode stmt successor)
            (link-dinode stmt (first (stmt-successors successor)))
            (altered-digraph stmtgraph))))
  (purge-unreachables-digraph stmtgraph))


;;;;+---------------------------+
;;;;| Split primitive operations. |
;;;;+---------------------------+
;;;Split primitive operations such as additions and multiplications of more than
;;;two arguments into sequences of primitive operations of two arguments.
(defun split-primitives (stmtgraph)
  (labels
    ((split-statement (stmt)
      (if (and
           (eq (stmt-operation stmt) 'primitive)
           (find (stmt-method stmt) '(+ * max min and or xor logand logor logxor))
           (> (length (stmt-args stmt)) 2))
        (let* ((new-var (gen-var stmtgraph))
               (new-stmt (make-stmt :operation 'primitive
                                    :method (stmt-method stmt)
                                    :target (stmt-target stmt)
                                    :args (cons new-var (cddr (stmt-args stmt))))))
          (setf (stmt-args stmt) (list (stmt-arg stmt) (stmt-arg2 stmt)))
          (setf (stmt-target stmt) new-var)
          (insert-dinode new-stmt stmt (first (stmt-successors stmt)))
          (altered-digraph stmtgraph)
          (split-statement new-stmt)))))
    (loop for stmt being the dinodes of stmtgraph do (split-statement stmt))))


;;;Transform CSENDs to instance variables into CSENDs to variables followed by MOVEs into
;;;the instance variables.
(defun transform-ivar-sends (stmtgraph)
  (loop for stmt being the dinodes of stmtgraph do
        (if (and
             (send-operation? (stmt-operation stmt))
             (ivar? (stmt-target stmt)))
          (let* ((new-var (gen-var stmtgraph))
                 (new-stmt (make-stmt :operation 'move
                                      :target (stmt-target stmt)
                                      :args (list new-var))))
            (setf (stmt-target stmt) new-var)
            (insert-dinode new-stmt stmt (first (stmt-successors stmt)))
            (altered-digraph stmtgraph)))))


;;;;+-------------------------------+
;;;;| Optimize primitive operations. |
;;;;+-------------------------------+

(defun optimize-primitive (stmt)
  (labels

    ;;Attempt to convert * to a logical shift.  Return true if successful.
    ((attempt-*-convert (arg1 arg2)
```

```
      (if (const? arg1)
        (let ((word (slot-num arg1)))
          (if (integer-word? word)
            (let ((value (data word)))
              (when (power-of-2? value)
                (setf (stmt-method stmt) 'ash)
                (setf (stmt-args stmt) (list arg2 (make-const (log2 value))))
                t)))))))

    (let ((arg1 (stmt-arg stmt))
          (arg2 (stmt-arg2 stmt)))
      (case (stmt-method stmt)
        (* (or (attempt-*-convert arg1 arg2)
               (attempt-*-convert arg2 arg1)))
        (// (if (const? arg2)
              (let ((word (slot-num arg2)))
                (if (integer-word? word)
                  (let ((value (data word)))
                    (when (power-of-2? value)
                      (setf (stmt-method stmt) 'ash)
                      (setf (stmt-args stmt) (list arg1 (make-const (- (log2 value))))))))))))))

;;;Optimize primitive operations for the idiosyncrasies of the MDP architecture.
;;;Whenever possible put long constants on the left sides of binary operations
;;;and short constants on the right sides.  Transform multiplications by powers
;;;of two into shifts.
(defun optimize-primitives (stmtgraph)
  (loop for stmt being the dinodes of stmtgraph do
        (if (eq (stmt-operation stmt) 'primitive)
          (optimize-primitive stmt))))



;;;;+--------------------------+
;;;;| Stmtgraph optimizations. |
;;;;+--------------------------+
;;;Perform iterative stmtgraph optimizations until a steady state is reached.
;;;Return the stmtgraph.
(defun iterative-optimize-stmtgraph (stmtgraph)
  (attribute-steady-state
   (stmtgraph-attributes stmtgraph)
   (progn
     (when *delete-dead-defs* (delete-dead-defs stmtgraph))
     (when *delete-moves* (delete-moves stmtgraph))
     (when *delete-touches* (delete-touches stmtgraph))
     (when *dflow-optimizations* (calc-dflow stmtgraph))
     (when *fold-constants* (fold-constants stmtgraph))
     (when *forward-sends* (forward-sends stmtgraph))
     (fold-conditionals stmtgraph) ;This must not be disabled, or code generator will fail!
     (when *merge-code*
       (merge-joins stmtgraph)
       (merge-forks stmtgraph))))
  stmtgraph)


;;;Perform stmtgraph post-optimizations and transformations.
;;;Return the stmtgraph.
(defun transform-stmtgraph (stmtgraph)
  (split-primitives stmtgraph)
  (transform-ivar-sends stmtgraph)
  (if *optimize-primitives* (optimize-primitives stmtgraph))
  stmtgraph)


;;;Perform all stmtgraph optimizations and transformations.
;;;Return the stmtgraph.
(defun optimize-stmtgraph (stmtgraph)
  (transform-stmtgraph (iterative-optimize-stmtgraph stmtgraph)))
```

Instruction Generator

```
;;;
```

```
;;;(stmt-used-regs stmt) values for each statement.
(defun calc-reg-requirements (stmtgraph)
  (loop for stmt being the dinodes of stmtgraph do
        (multiple-value-bind (n-unused-regs reserved-regs) (calc-stmt-reg-requirements stmt)
          (setf (stmt-n-unused-regs stmt) n-unused-regs)
          (setf (stmt-reserved-regs stmt) reserved-regs)
          (setf (stmt-used-regs stmt) b0))))

;;;Assign as many variables present in the vars-to-allocate bmap to registers as possible.
;;;Return a bmap of all variables that were successfully assigned to registers.
(defun assign-regs (stmtgraph varinfo vars-to-allocate)
  (calc-live stmtgraph)
  (calc-reg-requirements stmtgraph)
  (let ((assigned-vars b0)
        (priority-list
         (loop for var being the bits of vars-to-allocate collect
               (loop for stmt being the dinodes of stmtgraph
                     count (btest var (stmt-live-in stmt)) into nlive
                     count (var-referenced? (make-slot 'var var) stmt) into nrefs
                     finally (return (cons var (/ (float nrefs) (max nlive nrefs)))))))))
    (sort priority-list #'> :key #'cdr)
    (dolist (var-pair priority-list assigned-vars)
      (let ((var (car var-pair))
            (possible-regs '#((0 3))))
        (if
          (loop for stmt being the dinodes of stmtgraph do
                (when (btest var (stmt-live-in stmt))
                  (if (zerop (stmt-n-unused-regs stmt))
                    (return nil)
                    (setq possible-regs (b- possible-regs (b+ (stmt-reserved-regs stmt)
                                                              (stmt-used-regs stmt)))))
                  (if (bempty possible-regs)
                    (return nil)))
                finally (return t))
          (let ((the-reg (blow possible-regs)))
            (loop for stmt being the dinodes of stmtgraph do
                  (when (btest var (stmt-live-in stmt))
                    (decf (stmt-n-unused-regs stmt))
                    (bsetf (stmt-used-regs stmt) the-reg)))
            (setf (svref (varinfo-varlocs varinfo) var) (make-loc 'reg the-reg))
            (bsetf assigned-vars var)))))))

;;;;+----------------------------------+
;;;;| Allocate locations to variables. |
;;;;+----------------------------------+

;;;Find a coloring for the graph given by the edge-matrix.
;;; Edge-matrix must be a square, symmetric, bit matrix in which bit [i,j] is
;;; set iff there is an edge between nodes i and j.
;;; Nodes is a bmap with the bits set for the nodes in edge-matrix which are
;;; to be considered for coloring.  The other nodes are ignored.
;;;Return an array giving the color assignment of each node between 0 and
;;;(1- (array-dimension edge-matrix 0)) and the number of colors used to make the
;;;assignment.  The nodes specified in the nodes bmap are assigned colors such that
;;;no two nodes connected by an edge are assigned the same color.  The colors are
;;;positive integers starting at 0.  The nodes not specified in the nodes bmap are
;;;assigned nil colors.
(defun color-graph (edge-matrix nodes)
  (let* ((dim (array-dimension edge-matrix 0))
         ;;Edge-counts gives the number of edges left for each node.
         (edge-counts (make-array (list dim) :element-type 'integer))
         (assignments (make-array (list dim) :initial-element nil)))
    (labels

      ((color-graph-rec (nodes)
        (if (bempty nodes)
          0
          (let ((min-count -1)
                index)
            (loop for i being the bits of nodes do
                  (when (> (svref edge-counts i) min-count)
                    (setf min-count (svref edge-counts i))
                    (setf index i)))
            (let ((new-nodes (bclr index nodes)))
              (loop for i being the bits of new-nodes do
                    (if (= (bit edge-matrix i index) 1)
                      (decf (svref edge-counts i))))
              (let ((ncolors (color-graph-rec new-nodes))
                    (interfering-colors b0))
                (loop for i being the bits of new-nodes do
```

```
                    (if (= (bit edge-matrix i index) 1)
                      (bsetf interfering-colors (svref assignments i))))
                (let ((new-color (blow (bnot interfering-colors))))
                  (setf (svref assignments index) new-color)
                  (max ncolors (1+ new-color)))))))))

      (loop for i being the bits of nodes do
            (setf (svref edge-counts i)
                  (loop for j being the bits of nodes
                        sum (bit edge-matrix i j))))
      (let ((ncolors (color-graph-rec nodes)))
        (values assignments ncolors)))))


;;;Assign variables to context locations in the vlocs array using a very simple approach.
;;;Return the number of locations used.
(defun simple-assign-vlocs (stmtgraph varinfo)
  (dotimes (var (stmtgraph-nvars stmtgraph))
    (unless (svref (varinfo-varlocs varinfo) var)
      (setf (svref (varinfo-varlocs varinfo) var)
            (make-loc 'vloc (+ var first-context-slot-num)))))
  (stmtgraph-nvars stmtgraph))


;;;Assign variables to context locations in the vlocs array by calculating a variable
;;;interference graph and trying to pack as many variables into as few locations as possible.
;;;Return the number of locations used.
(defun alloc-vlocs (stmtgraph varinfo vars-to-allocate)
  (calc-live stmtgraph)
  (calc-waiting stmtgraph)
  (let ((interference-matrix (make-array (list (stmtgraph-nvars stmtgraph)
                                               (stmtgraph-nvars stmtgraph))
                                         :element-type 'bit
                                         :initial-element 0)))
    (loop for stmt being the dinodes of stmtgraph do
          (let ((target (stmt-target stmt)))
            (if (var? target)
              (let ((d (slot-num target)))
                (loop for i being the bits of (b* (bclr d vars-to-allocate)
                                                  (b+ (stmt-live-out stmt)
                                                      (stmt-waiting-out stmt))) do
                      (progn
                        (setf (bit interference-matrix i d) 1)
                        (setf (bit interference-matrix d i) 1)))))))
    (multiple-value-bind (assignments ncolors) (color-graph interference-matrix
                                                            vars-to-allocate)
      (loop for i being the bits of vars-to-allocate do
            (setf (svref (varinfo-varlocs varinfo) i)
                  (make-loc 'vloc (+ (svref assignments i) first-context-slot-num))))
      ncolors)))


;;;;+------------------------------+
;;;;| Calculate the varinfo record. |
;;;;+------------------------------+
;;;Return the varinfo record.  Modify the stmtgraph as appropriate.
(defun new-varinfo (stmtgraph nargs nivars)
  (let ((varinfo (make-varinfo
                  :nargs nargs
                  :nivars nivars
                  :ivars-used (if *optimize-ivar-access*
                                  (referenced-ivars? stmtgraph)
                                  nivars)
                  :varlocs (make-array (list (stmtgraph-nvars stmtgraph)) :initial-element nil)))
        (vars-to-allocate (referenced-vars stmtgraph)))
    (if *reg-variables*
      (setq vars-to-allocate (b- vars-to-allocate
                                 (assign-regs stmtgraph varinfo
                                              (b- vars-to-allocate (target-vars stmtgraph))))))
    (let ((nvlocs (if *optimize-vars*
                    (alloc-vlocs stmtgraph varinfo vars-to-allocate)
                    (simple-assign-vlocs stmtgraph varinfo))))
      (if (> nvlocs (- max-context-size first-context-slot-num))
        (cerror "Compile anyway" "Too many local variables and temporaries: ~D (~D maximum)"
                nvlocs (- max-context-size first-context-slot-num)))
      (if (and *optimize-null-contexts* (zerop nvlocs))
        (setq nvlocs nil))
      (setf (varinfo-nvlocs varinfo) nvlocs))
    varinfo))
```

```
;;;;+----------------+
;;;;| Frame routines |
;;;;+----------------+
(defstruct (frame (:copier copy-frame1))
  varinfo                  ;Global varinfo assignments.
  (regs (make-array '(4))) ;Array of known register slot values.
  (lockmap b0 :type bmap)  ;Bmap of register locks.
  (waiting b0 :type bmap)  ;Bmap of unforced slots.
  (migrate t)              ;True if the instance object could have migrated away.
  (lru-regs '(0 1 2 3)))   ;List of registers in order from most to least recently used.

(defvar *frame*)

;;;Bring the register reg to the front of lru-regs.
;;;Return the new lru-regs.
(defun bring-to-front (reg lru-regs)
  (if *use-lru-register-allocation* (cons reg (remove reg lru-regs))
      lru-regs))


;;;Add a temporary binding of slot to reg in frame.
(defun add-temp-location (reg slot)
  (if slot (push slot (svref (frame-regs *frame*) (reg-num reg)))))

;;;Add a temporary binding of slot to reg in frame.
;;;Purge all existing temporary bindings referring to reg in frame.
(defun new-temp-location (reg slot)
  (setf (svref (frame-regs *frame*) (reg-num reg)) (and slot (list slot))))

;;;Purge all temporary bindings referring to reg in frame.
(defun trash-reg (reg)
  (setf (svref (frame-regs *frame*) (reg-num reg)) nil))

;;;Purge all temporary bindings of slot in frame.
(defun purge-temp-locations (slot)
  (dotimes (r 4)
    (setf (svref (frame-regs *frame*) r)
          (delete slot (svref (frame-regs *frame*) r) :test #'equal))))

;;;Purge all temporary bindings referring to ilocs in frame.
(defun purge-iloc-locations ()
  (dotimes (r 4)
    (setf (svref (frame-regs *frame*) r)
          (delete-if #'(lambda (slot) (ivar? slot)) (svref (frame-regs *frame*) r)))))

;;;Return the permanent location of slot in frame.
(defun the-location (slot)
  (let ((num (slot-num slot)))
    (case (slottype slot)
      (var (svref (varinfo-varlocs (frame-varinfo *frame*)) num))
      (arg (make-loc 'aloc (+ num first-arg-slot-num)))
      (ivar (make-loc 'iloc (+ num first-instance-slot-num)))
      (const (make-loc (if (short-word? (slot-num slot)) 'sconst 'lconst) num))
      (self selfloc)
      (loc num)
      (t (error "Bad slot: ~S" slot)))))

;;;Return true if the register contains the value of the slot in the frame.
(defun reg-contains (reg slot)
  (and *optimize-local-regs*
       (find slot (svref (frame-regs *frame*) (reg-num reg)) :test #'equal)))

;;;Return a list of all locations referring to slot in frame.
(defun locations (slot)
  (let ((loclist (list (the-location slot))))
    (if *optimize-local-regs*
      (dotimes (r 4)
        (if (find slot (svref (frame-regs *frame*) r) :test #'equal)
          (push (make-loc 'reg r) loclist))))
    loclist))

;;;Allocate a temporary register that is not one of the forbidden registers.
(defun alloc-reg (&key (forbidden b0))
  (let ((bad (b+ (frame-lockmap *frame*) forbidden))
        excellent
        good)
    (dolist (r (frame-lru-regs *frame*))
      (unless (btest r bad)
        (if (or (zerop r) (svref (frame-regs *frame*) r))
          (setq good r)
          (setq excellent r))))
```

```
    (let ((reg (or excellent good)))
      (if reg
        (progn
          (setf (frame-lru-regs *frame*) (bring-to-front reg (frame-lru-regs *frame*)))
          (make-loc 'reg reg))
        (error "Can't allocate register")))))

;;;Lock the register reg, which prevents it from being allocated until it is
;;;unlocked.
(defun lock-reg (reg)
  (bsetf (frame-lockmap *frame*) (reg-num reg)))

;;;Unlock the register.  Do nothing if it was previously unlocked.
(defun unlock-reg (reg)
  (bclrf (frame-lockmap *frame*) (reg-num reg)))

;;;Mark the slot as being unforced.
(defun unforce (slot)
  (and (var? slot)
       (bsetf (frame-waiting *frame*) (slot-num slot))))

;;;Mark the slot as being forced.
(defun force (slot)
  (and (var? slot)
       (bclrf (frame-waiting *frame*) (slot-num slot))))

;;;Return true if slot is a potentially unforced context variable.
(defun unforced (slot)
  (and (var? slot)
       (or (not *optimize-forces*)
           (btest (slot-num slot) (frame-waiting *frame*)))))

;;;Return a list of all potentially unforced context variable slots.
(defun unforced-slots ()
  (let ((varlocs (varinfo-varlocs (frame-varinfo *frame*))))
    (loop for i below (length varlocs)
          if (and (vloc? (svref varlocs i))
                  (unforced (make-slot 'var i)))
          collect (make-slot 'var i))))

;;;Assert that the instance object could have migrated away.
(defun could-migrate ()
  (purge-iloc-locations)
  (setf (frame-migrate *frame*) t))

;;;Assert that the instance object could not have migrated away.
(defun could-not-migrate ()
  (setf (frame-migrate *frame*) nil))

;;;Return true if the instance object could have migrated away.
(defun migratep ()
  (or (not *optimize-migrate*)
      (frame-migrate *frame*)))

;;;Return the number of variable locations in context.
(defmacro get-nvlocs ()
  '(varinfo-nvlocs (frame-varinfo *frame*)))

;;;Return the size of the message that started this method.
(defun get-msgsize ()
  (+ msg-overhead 1 (varinfo-nargs (frame-varinfo *frame*))))

;;;Return the number of instance variables or nil if the class is atomic.
(defmacro get-nivars ()
  '(varinfo-nivars (frame-varinfo *frame*)))

;;;Return true if instance variables are used.
(defun ivars-used? ()
  (varinfo-ivars-used (frame-varinfo *frame*)))

(eval-when (compile load eval)
  (if *reply-node*
    (progn
      ;;;Return the reply ID.
      (defun reply-ID ()
        (make-loc 'aloc (- (get-msgsize) 3)))
      ;;;Return the reply slot.
      (defun reply-slot ()
        (make-loc 'aloc (- (get-msgsize) 2)))
      ;;;Return the reply node number.
      (defun reply-node ()
```

```
        (make-loc 'aloc (- (get-msgsize) 1))))
    (progn
      ;;;Return the reply ID.
      (defun reply-ID ()
        (make-loc 'aloc (- (get-msgsize) 2)))
      ;;;Return the reply slot.
      (defun reply-slot ()
        (make-loc 'aloc (- (get-msgsize) 1))))))


;;;Set the register values in regs1 to be the intersection of the values in regs1
;;;and regs2.
(defun merge-frame-regs (regs1 regs2)
  (dotimes (r 4)
    (setf (svref regs1 r)
          (copy-list (nintersection (svref regs1 r)
                                    (svref regs2 r) :test #'equal)))))

;;;Merge the two frames to produce a frame containing the common bindings of
;;;the two frames.  If either frame is nil, a frame equivalent to the other frame
;;;is returned.  Frame1 may be destructively modified.  The resulting frame does not
;;;have any common elements with frame2.
(defun merge-frames (frame1 frame2)
  (merge-frame-regs (frame-regs frame1) (frame-regs frame2))
  (setf (frame-lockmap frame1) (b* (frame-lockmap frame1) (frame-lockmap frame2)))
  (setf (frame-waiting frame1) (b+ (frame-waiting frame1) (frame-waiting frame2)))
  (setf (frame-migrate frame1) (or (frame-migrate frame1) (frame-migrate frame2)))
  frame1)

;;;Merge the frame into *frame*.
(defun merge-frame (frame)
  (setq *frame* (merge-frames *frame* frame)))

;;;Make a copy of the registers array.
(defun copy-frame-regs (regs)
  (map '(simple-vector 4) #'copy-list regs))

;;;Make a copy of the frame.
(defun copy-frame (frame)
  (let ((copy (copy-frame1 frame)))
    (setf (frame-regs copy) (copy-frame-regs (frame-regs frame)))
    copy))

;;;Return a copy of *frame*.
(defun current-frame ()
  (copy-frame *frame*))


;;;Calculate a frame for the statement from it and its predecessors.
(defun gen-frame (stmt varinfo)
  (let ((frame (make-frame
                :varinfo varinfo
                :regs nil
                :lockmap (if *reg-variables*
                             (stmt-used-regs stmt)
                             b0)
                :waiting (if *optimize-forces*
                             (stmt-waiting-in stmt)
                             b1)
                :migrate nil)))
    (dolist (predecessor (stmt-predecessors stmt))
      (if (or (root? predecessor) (null (stmt-frame predecessor)))
          (progn
            (setf (frame-regs frame) (make-array '(4)))
            (setf (frame-migrate frame) t))
          (let ((pred-frame (stmt-frame predecessor)))
            (setf (frame-lru-regs frame) (frame-lru-regs pred-frame))
            (if (frame-regs frame)
                (merge-frame-regs (frame-regs frame) (frame-regs pred-frame))
                (setf (frame-regs frame) (copy-frame-regs (frame-regs pred-frame))))
            (if (frame-migrate pred-frame)
                (setf (frame-migrate frame) t)))))
    (unless (in-same-basic-block? (car (stmt-predecessors stmt)) stmt)
      (setf (svref (frame-regs frame) 0) nil))
    frame))


;;;Return the "best" location out of the given list that is not one of the
;;;registers indicated in the forbidden bmap.
(defun best-loc (loc-list &key (forbidden b0))
  (let (excellent very-good good poor)
```

```
    (dolist (loc loc-list)
      (cond ((reg? loc)
             (unless (btest (reg-num loc) forbidden)
               (if (btest (loc-num loc) (frame-lockmap *frame*))
                 (setq excellent loc) ;Prefer locked registers to unlocked ones.
                 (setq very-good loc))))
            ((sconst? loc) (setq good loc))
            (t (setq poor loc))))
    (or excellent very-good good poor)))


;;;;+--------------------------------+
;;;;| Instruction generator routines |
;;;;+--------------------------------+
(defvar *first-inst*)
(defvar *last-insts*)
(defvar *prev*) ;Previous instruction in code sequence.

;;;Call f to generate instructions under the given frame.
;;;Return three values:
;;; the first instruction,
;;; a list of last instructions,
;;; and the frame.
(defun gen-insts (f frame)
  (let ((*first-inst* nil)
        (*last-insts* nil)
        (*frame* frame))
    (funcall f)
    (values *first-inst* *last-insts* *frame*)))

;;;Link the instruction to the end of the generated instructions.
;;;Return inst.
(defun gen (inst)
  ;;Create static links.
  (link-inst *prev* inst)
  (setq *prev* inst)
  ;;Create dynamic links.
  (dolist (last *last-insts*) (link-dinode last inst))
  (if (null *first-inst*) (setq *first-inst* inst))
  (setq *last-insts* (list inst))
  inst)

;;;Generate the instruction using arguments as inputs to new-inst.
;;;Return the instruction.
(defmacro gen-inst (&rest arguments)
  (list 'gen (cons 'new-inst arguments)))

;;;Indicate that the next instruction should not be linked to the current one.
(defun gen-break ()
  (setq *last-insts* nil))

;;;Indicate that the instruction given as an argument is a branch to the current
;;;position.  It will be linked to the next instruction generated.
(defun gen-merge (inst)
  (push inst *last-insts*))


;;;Generate an instruction to read src into dst, which must be a register.
;;;If src is a lconst, dst must be reg0.
(defun gen-read (srcloc srcslot dstreg)
  (unless (or (equal srcloc dstreg) (reg-contains dstreg srcslot))
    (if (lconst? srcloc)
      (progn
        (assert (equal dstreg reg0))
        (gen-inst :op 'dc :srcl srcloc :writes '#(0)))
      (gen-inst :op 'move :srcl srcloc :dst dstreg)))
  (new-temp-location dstreg srcslot)
  (if (iloc? srcloc) (could-not-migrate))
  (when (unforced srcslot)
    (could-migrate)
    (force srcslot)))

;;;Generate an instruction to read a constant or a special register.  The fact
;;;that the constant or special register was read into dstreg is kept in the frame until
;;;dstreg is altered, allowing elimination of subsequent reads of the same constant or
;;;special register into the same dstreg.  The constant or special register must therefore
;;;be immutable.
(defun gen-read-special (srcloc dstreg)
  (gen-read
   srcloc
```

```
   (case (loctype srcloc)
     ((sconst lconst) (make-slot 'const (loc-num srcloc)))
     (t (make-slot 'loc srcloc)))
   dstreg))

;;;Generate instruction(s) to read the value in srcslot into dstreg.  The read is
;;;direct except when srcslot is a lconst, in which case it is read into reg0 first
;;;if can-trash-reg0 is true and an error is generated otherwise.
(defun gen-read-slot (srcslot dstreg &optional can-trash-reg0)
  (let ((srcloc (best-loc (locations srcslot))))
    (if (and can-trash-reg0 (lconst? srcloc) (not (equal dstreg reg0)))
      (progn
        (gen-read srcloc srcslot reg0)
        (gen-read reg0 srcslot dstreg))
      (gen-read srcloc srcslot dstreg))))

;;;Generate an instruction to write src, which must be a register, to dst.
(defun gen-write (srcreg dstloc dstslot)
  (unless (equal srcreg dstloc)
    (gen-inst
     :op 'move
     :src1 srcreg
     :dst dstloc))
  (if (iloc? dstloc) (could-not-migrate))
  (purge-temp-locations dstslot)
  (if (reg? dstloc) (new-temp-location dstloc dstslot))
  (add-temp-location srcreg dstslot))

;;;Generate an instruction to make a system call.
;;;Reads and writes are optional parameters that specify the registers that the
;;;system call reads and writes/trashes.
(defun gen-call (trapnum &key (reads '#{}) (writes '#{}))
  (could-migrate)
  (gen-inst :op 'call :src1 (make-loc 'sconst trapnum) :reads reads :writes writes)
  (loop for r being the bits of writes do (trash-reg (make-loc 'reg r))))


;;;;+-----------------------+
;;;;| Generate an instgraph. |
;;;;+-----------------------+

;;;Linearize and convert the stmtgraph into a module containing an MDP instruction digraph.
;;;Return the module.
(defun compile-stmtgraph (stmtgraph nargs nivars)
  (gen-stmtgraph-insts stmtgraph (new-varinfo stmtgraph nargs nivars)))

;;;Linearize and convert the stmtgraph and varinfo into a module containing an
;;;MDP instruction digraph.
;;;Return the module.
(defun gen-stmtgraph-insts (stmtgraph varinfo)
  (let* ((m (make-module))
         (*prev* m))
    (setf (module-digraph m)
          (map-digraph
           stmtgraph
           #'(lambda (stmt) (gen-frame-insts stmt varinfo))
           :order (linearize stmtgraph)))
    (link-inst *prev* m) ;Close the loop.
    m))

;;; Returns the instructions generated for stmt.
(defun gen-frame-insts (stmt varinfo)
  (multiple-value-bind (first last frame)
                       (gen-insts #'(lambda () (gen-stmt-insts stmt))
                                  (gen-frame stmt varinfo))
    (setf (stmt-frame stmt) frame)
    (values first last)))


;;; Returns the instructions generated for stmt starting from *frame*.
;;; In this and all following procedures *frame* is modified to reflect
;;; the state at the end of the statement.
(defun gen-stmt-insts (stmt)
  (ecase (stmt-operation stmt)
         (enter (gen-enter-insts stmt))
         ((csend rsend) (gen-send-insts stmt))
         (primitive (gen-primitive-insts stmt))
         (touch (gen-touch-insts stmt))
         (move (gen-move-insts stmt))
         (new (gen-new-insts stmt))
         (condition (gen-cond-insts stmt))
```

```
        (reply (gen-reply-insts stmt))
        (exit (gen-exit-insts stmt))))


(defun gen-move-insts (stmt)
  (unless (equal (stmt-target stmt) (stmt-arg stmt))
    (gen-force (stmt-target stmt))) ;Avoid putting target into limbo.
  (let* ((srcslot (stmt-arg stmt))
         (srclocs (locations srcslot))
         (srcloc (best-loc srclocs))
         (dstslot (stmt-target stmt))
         (dstloc (the-location dstslot)))
    (unless (find dstloc srclocs :test #'equal)
      (case (loctype dstloc)
        (reg
         (gen-read-slot srcslot dstloc t)
         (purge-temp-locations dstslot))
        ((vloc iloc)
         (case (loctype srcloc)
           (reg
            (gen-write srcloc dstloc dstslot))
           ((sconst lconst vloc aloc iloc)
            (gen-read srcloc srcslot reg0)
            (gen-write reg0 dstloc dstslot)))))))
  ;;Indicate that all source registers contain the destination value too.
  (dolist (srcloc (locations (stmt-arg stmt)))
    (if (reg? srcloc) (add-temp-location srcloc (stmt-target stmt)))))

(defun gen-primitive-insts (stmt)
  (unless (find (stmt-target stmt) (stmt-args stmt) :test #'equal)
    (gen-force (stmt-target stmt))) ;Avoid putting target into limbo.
  (let ((method (stmt-method stmt)))
    (unless
      (case (length (stmt-args stmt))
        (1 (let ((unary-name (cadr (assoc method
                                          '((neg neg)
                                            (not not)
                                            (lognot not))))))
             (cond
              (unary-name (gen-unary-primitive-insts unary-name stmt) t))))
        (2 (let ((binary-name (cadr (assoc method
                                           '((and and)
                                             (or or)
                                             (xor xor)
                                             (logand and)
                                             (logor or)
                                             (logxor xor)
                                             (+ add)
                                             (- sub)
                                             (* mul)
                                             (max max)
                                             (min min)
                                             (ash ash)
                                             (< lt)
                                             (<= le)
                                             (> gt)
                                             (>= ge)
                                             (= equal)
                                             (<> nequal)
                                             (eq eq)
                                             (neq neq))))))
             (cond
              (binary-name (gen-binary-primitive-insts binary-name stmt) t)
              ((eq method '//) (gen-divide-insts stmt nil) t)
              ((eq method 'mod) (gen-divide-insts stmt t) t)))))
      (error "Bad primitive: ~S ~S" method (stmt-args stmt)))))


;;; Allocate a register for the calculation of a value to be stored in dstslot.
;;; If dstslot is a register, it is used; otherwise, a new register is allocated.
;;; The function f should generate code to calculate the value for dstslot and
;;; store it in the register it receives as an argument.
;;; Alloc-dst-reg generates the code generated by f followed by code to move the
;;; resulting value to dstslot, if necessary.
(defun alloc-dst-reg (dstslot f)
  (let ((dstloc (the-location dstslot)))
    (case (loctype dstloc)
      (reg
       (funcall f dstloc)
       (purge-temp-locations dstslot)
       (trash-reg dstloc))
```

```
      ((vloc iloc)
       (let ((dstreg (alloc-reg)))
         (funcall f dstreg)
         (gen-write dstreg dstloc dstslot))))))

;;; Bring the value in srcslot into R0 if it cannot be accessed as op0 of an
;;; instruction.  This only happens if srcslot is a lconst.
;;; The function f should generate code to use the value in srcslot.
(defun alloc-src-loc (srcslot f)
  (let ((srcloc (best-loc (locations srcslot))))
    (if (lconst? srcloc)
      (progn (gen-read srcloc srcslot reg0)
             (funcall f reg0))
      (funcall f srcloc))))

;;; Bring the value in srcslot into a register that is not one of the forbidden registers
;;; specified in the forbidden bmap.
;;; R0 may also be modified (even if it is forbidden) if srcslot is a lconst.
;;; The function f should generate code to use the value in srcslot.
(defun alloc-src-reg (srcslot f &key (forbidden b0))
  (labels
    ((alloc-src-reg-sub (bestsrcloc srclocs)
      (let ((srcloc (best-loc srclocs :forbidden forbidden)))
        (if (reg? srcloc)
          (funcall f srcloc)
          (let ((srcreg (alloc-reg :forbidden forbidden)))
            (gen-read bestsrcloc srcslot srcreg)
            (funcall f srcreg))))))
    (let* ((srclocs (locations srcslot))
           (srcloc (best-loc srclocs)))
      (if (lconst? srcloc)
        (progn (gen-read srcloc srcslot reg0)
               (alloc-src-reg-sub srcloc '#.(list reg0)))
        (alloc-src-reg-sub srcloc srclocs)))))

;;; Generate the instructions for the unary statement stmt using the instruction
;;; in op.
(defun gen-unary-primitive-insts (op stmt)
  (alloc-src-loc
   (stmt-arg stmt)
   #'(lambda (srcloc)
       (alloc-dst-reg
        (stmt-target stmt)
        #'(lambda (dstloc)
            (could-migrate)
            (force (stmt-arg stmt))
            (gen-inst :op op :src1 srcloc :dst dstloc)
            (trash-reg dstloc))))))

;;;If reversing the operands of the instruction would make it more efficient, return the
;;;new opcode of the instruction.  Otherwise, return nil.
(defun optimize-binary-order (op arg1 arg2)
  (let ((converse (cadr (assoc op '((and and)
                                    (or or)
                                    (xor xor)
                                    (add add)
                                    (mul mul)
                                    (max max)
                                    (min min)
                                    (lt gt)
                                    (le ge)
                                    (gt lt)
                                    (ge le)
                                    (equal equal)
                                    (nequal nequal)
                                    (eq eq)
                                    (neq neq))))))
    (and converse
         (or (not (eq op 'mul))
             (integer-const? arg1)
             (integer-const? arg2))
         (let ((locl (best-loc (locations arg1)))
               (loc2 (best-loc (locations arg2))))
           (and (or (and (lconst? loc2) (not (lconst? locl)))
                    (and (reg? loc2) (not (reg? locl))))
                converse))))))

;;;Generate the instructions for the binary statement stmt using the instruction in op.
(defun gen-binary-primitive-insts (op stmt)
  (let* ((arg1 (stmt-arg stmt))
         (arg2 (stmt-arg2 stmt))
```

```
        (order (and *optimize-primitives* (optimize-binary-order op arg1 arg2))))
    (when order
      (setq op order)
      (psetq arg1 arg2 arg2 arg1))
    (alloc-src-reg
     arg1
     #'(lambda (src1loc)
         (alloc-src-loc
          arg2
          #'(lambda (src2loc)
              (alloc-dst-reg
               (stmt-target stmt)
               #'(lambda (dstloc)
                   (could-migrate)
                   (force arg1)
                   (force arg2)
                   (gen-inst :op op :src1 src1loc :src2 src2loc :dst dstloc)
                   (trash-reg dstloc))))))
     :forbidden (if (lconst? (best-loc (locations arg2))) '#(0) b0))))

;;;Generate the instructions for the divide and remainder statements.
(defun gen-divide-insts (stmt remainder?)
  (gen-read-slot (stmt-arg stmt) reg1 t)
  (gen-read-slot (stmt-arg2 stmt) reg0)
  (gen-call "Divide" :reads '#(0 1) :writes '#(0 1))
  (let ((target (stmt-target stmt)))
    (gen-write (if remainder? reg1 reg0) (the-location target) target)))


;;;Generate the conditional instructions.
(defun gen-cond-insts (stmt)
  (alloc-src-reg
   (stmt-arg stmt)
   #'(lambda (srcloc)
       (could-migrate)
       (force (stmt-arg stmt))
       (gen-inst :op (stmt-method stmt) :src1 srcloc :src2 '#.(make-loc 'rel)))))


;;;Generate code to force slot by reading it into a register.
;;;Do not use one of the forbidden registers.
(defun gen-force (slot &key (forbidden b0))
  (if (unforced slot)
    (gen-read (the-location slot) slot (alloc-reg :forbidden forbidden))))


;;;Generate code to touch a variable.
(defun gen-touch-insts (stmt)
  (gen-force (stmt-arg stmt)))


;;;Generate code to create a new object.
(defun gen-new-insts (stmt)
  (gen-force (stmt-target stmt)) ;Avoid putting target into limbo.
  (gen-read-special (make-sconst (+ (class-nivars (stmt-method stmt)) first-instance-slot-num)) reg1)
  (gen-read-special (make-lconst 'class (stmt-method stmt)) reg0)
  (gen-call "New_Object" :reads '#(0 1) :writes '#(0 1))
  (could-migrate)
  (let ((target (stmt-target stmt)))
    (gen-write reg0 (the-location target) target)))


;;;Generate the value return code.
(defun gen-reply-insts (stmt)
  (let* ((arg (stmt-arg stmt))
         (tempreg (alloc-reg :forbidden '#(0)))
         (forbidden-map (bset 0 (bset (reg-num tempreg)))))
    (if (and (ivar? arg) (migratep))
      (gen-read (the-location arg) arg (alloc-reg :forbidden forbidden-map))
      (gen-force arg :forbidden forbidden-map))
    (if *reply-node*
      (progn
        (gen-read-special (reply-slot) tempreg)
        (let* ((branch (gen-inst :op 'bnil :src1 tempreg :src2 '#.(make-loc 'rel)))
               (frame (current-frame)))
          (gen-read-special '#.(make-lconst tMSG '("ReplyConst" . 4)) reg0)
          (gen-inst :op 'send2 :src1 (reply-node) :src2 reg0)
          (gen-inst :op 'send2 :src1 (reply-ID) :src2 tempreg)
          (alloc-src-loc
           arg
           #'(lambda (srcloc) (gen-inst :op 'sende :src1 srcloc)))
```

```
          (merge-frame frame)
          (gen-merge branch)))
        (progn
          (gen-read-special (reply-ID) tempreg)
          (let* ((nodereg (alloc-reg :forbidden (bset (reg-num tempreg) '#(0))))
                 (branch (gen-inst :op 'bnil :src1 tempreg :src2 '#.(make-loc 'rel)))
                 (frame (current-frame)))
            (gen-read-special '#.(make-lconst tMSG '("ReplyConst" . 4)) reg0)
            (gen-inst :op 'wtag :src1 tempreg :src2 (make-sconst tINT) :dst nodereg)
            (gen-inst :op 'lsh :src1 nodereg :src2 (make-sconst -16) :dst nodereg)
            (gen-inst :op 'send2 :src1 nodereg :src2 reg0)
            (gen-inst :op 'send :src1 tempreg)
            (gen-inst :op 'send :src1 (reply-slot))
            (alloc-src-loc
             arg
             #'(lambda (srcloc) (gen-inst :op 'sende :src1 srcloc)))
            (merge-frame frame)
            (gen-merge branch))))))


;;;Generate the code to begin execution.
(defun gen-enter-insts (stmt)
  (declare (ignore stmt))
  (when (get-nvlocs)
    (gen-read-special (make-sconst (get-nvlocs)) reg0)
    (gen-call "New_Context" :reads '#(0) :writes '#(0 1)))
  (gen-inst :op 'init-vlocs :writes '#(0))
  (when (ivars-used?)
    (gen-read selfloc '#.(make-slot 'self) reg0)
    (gen-inst :op 'xlate
              :src1 reg0
              :src2 '#.(make-sconst tINT "XLATE_OBJ")
              :dst instance-a-reg)
    (could-not-migrate)))


;;;Generate the code to terminate execution.
(defun gen-exit-insts (stmt)
  (declare (ignore stmt))
  (when (get-nvlocs)
    (let ((forced-locs b0))
      (dolist (slot (unforced-slots))
        (let ((loc (the-location slot)))
          (unless (btest (loc-num loc) forced-locs)
            (bsetf forced-locs (loc-num loc))
            (gen-force slot)))))
    (gen-call "Free_Context" :writes '#(0 1)))
  (gen-inst :op 'suspend)
  (gen-break))


;;;Generate code to do a csend or a rsend.
(defun gen-send-insts (stmt)
  (labels
    ((force-instobj (args tempreg)
      (cond
       ((endp args) nil)
       ((ivar? (car args))
        (gen-read (the-location (car args)) (car args) tempreg))
       (t (force-instobj (cdr args) tempreg)))))

    (gen-force (stmt-target stmt)) ;Avoid putting target into limbo.
    (let ((receiver (if (stmt-method stmt) (stmt-arg stmt) (stmt-arg2 stmt))))
      (dolist (arg (reverse (stmt-args stmt)))
        (unless (equal arg receiver)
          (gen-force arg))) ;Force all arguments except the receiver.
      (if (and *optimize-send-self* (self? receiver) (get-nivars))
        (progn
          (if (migratep)
            (force-instobj (stmt-args stmt) reg1)) ;Force instance object if necessary.
          (gen-read-special '#.(make-loc 'sreg 'nnr) reg1))
        (progn
          (gen-read-slot receiver reg0)
          (if (migratep)
            (force-instobj (stmt-args stmt) reg1)) ;Force instance object if necessary.
          (gen-call "Send_Node_Nr" :reads '#(0) :writes '#(0 1)))))
    (gen-read-special
     (make-lconst tMSG (cons "SendConst" (+ 4
                                            (if (stmt-method stmt) 1 0)
                                            (length (stmt-args stmt)))))
     reg0)
```

```
(gen-inst :op 'send2 :src1 reg1 :src2 reg0)
(when (stmt-method stmt)
  (gen-read-special (make-lconst 'method (stmt-method stmt)) reg0)
  (gen-inst :op 'send :src1 reg0))
(dolist (arg (stmt-args stmt))
  (alloc-src-loc
   arg
   #'(lambda (argloc)
       (gen-inst :op 'send :src1 argloc))))
(cond
 ((eq (stmt-operation stmt) 'rsend)
  (gen-inst :op 'send :src1 (reply-ID))
  (if *reply-node*
    (progn
      (gen-inst :op 'send :src1 (reply-slot))
      (gen-inst :op 'sende :src1 (reply-node)))
    (gen-inst :op 'sende :src1 (reply-slot))))
 ((null (stmt-target stmt))
  (gen-inst :op 'send :src1 '#.(make-loc 'sconst wNIL))
  (if *reply-node* (gen-inst :op 'send :src1 '#.(make-loc 'sconst wNIL)))
  (gen-inst :op 'sende :src1 '#.(make-loc 'sconst wNIL)))
 (t
  (let ((dstloc (the-location (stmt-target stmt))))
    (gen-inst :op 'send :src1 contextID)
    (assert (vloc? dstloc))
    (if *reply-node*
      (progn
        (gen-read-special '#.(make-loc 'sreg 'nnr) reg1)
        (gen-inst :op 'send2e
                  :src1 (make-sconst (loc-num dstloc))
                  :src2 reg1))
      (gen-inst :op 'sende :src1 (make-sconst (loc-num dstloc))))
    (gen-inst :op 'wtag :src1 reg1 :src2 '#.(make-sconst tCFUT) :dst reg1)
    (gen-write reg1 dstloc (stmt-target stmt))
    (trash-reg reg1)
    (unforce (stmt-target stmt))))))
```

Assembly Code Generator

```
;;;
```

```
   'vlive (digraph-attributes (module-digraph module))
   (macro-relax (module-digraph module)
                #'inst-vlive
                #'(lambda (inst new-vlive) (setf (inst-vlive inst) new-vlive))
                #'(lambda (inst)
                    (let ((vlive (inst-vlive inst)))
                      (if (vloc? (inst-dst inst))
                        (bclrf vlive (loc-num (inst-dst inst))))
                      (if (vloc? (inst-srcl inst))
                        (bsetf vlive (loc-num (inst-srcl inst))))
                      (if (vloc? (inst-src2 inst))
                        (bsetf vlive (loc-num (inst-src2 inst))))
                      vlive))
                :from-end t)))


;;;;+--------------------------------------+
;;;;| Initialize variables where necessary. |
;;;;+--------------------------------------+
(defun init-vlocs (module)
  (calc-live-vlocs module)
  (do ((inst (module-next module) (inst-next inst)))
      ((module? inst) (error "Can't find INIT-VLOCS"))
    (when (eq (inst-op inst) 'init-vlocs)
      (let ((vlive (b- (inst-vlive inst) '#.(brange first-context-slot-num)))
            (next (inst-next inst)))
        (if (bempty vlive)
          (delete-module module inst)
          (progn
            (setf (inst-op inst) 'move)
            (setf (inst-srcl inst) '#.(make-loc 'sconst wNIL))
            (setf (inst-dst inst) reg0)
            (loop for locnum being the bits of vlive do
                  (insert-module module
                                 (new-inst :op 'move
                                           :srcl reg0
                                           :dst (make-loc 'vloc locnum))
                                 (inst-prev next)
                                 next)))))
      (return))))
```

```
;;;;+--------------------------------------+
;;;;| Test whether two instructions commute. |
;;;;+--------------------------------------+

;;;Return true if the two instructions inst1 and inst2 could be transposed without changing
;;;the meaning of the program.
;;;####This routine is not perfect--it does not catch some of the dependencies involving
;;;special registers, but it does work in the simple cases in which it is called.####
(defun insts-commute? (inst1 inst2)
  (flet ((same? (val1 val2) (and val1 (equal val1 val2))))
    (let ((op1 (inst-op inst1))
          (op2 (inst-op inst2)))
      (not (or (branch? op1)
               (branch? op2)
               (eq op1 'suspend)
               (eq op1 'res)
               (eq op2 'suspend)
               (eq op2 'res)
               (not (bempty (b* (b+ (inst-reads inst1) (inst-reads inst2))
                                (b+ (inst-writes inst1) (inst-writes inst2)))))
               (same? (inst-srcl inst1) (inst-dst inst2))
               (same? (inst-src2 inst1) (inst-dst inst2))
               (same? (inst-srcl inst2) (inst-dst inst1))
               (same? (inst-src2 inst2) (inst-dst inst1))
               (same? (inst-dst inst1) (inst-dst inst2))
               (areg? (inst-dst inst1))
               (areg? (inst-dst inst2))
               (sreg? (inst-dst inst1))
               (sreg? (inst-dst inst2))
               (and (send-op? op1) (send-op? op2))
               (and (stack-op? op1) (stack-op? op2))
               (and (assoc-op? op1) (assoc-op? op2)))))))


;;;;+----------------------------------------+
;;;;| Calculate the pc values and compact DC's. |
;;;;+----------------------------------------+

;;;Return the last instruction before inst in the same basic block as inst that
```

```
;;;satisfies test.  Return NIL if there is no such instruction.
(defun prev-inst-that (inst test)
  (do ((i (inst-prev inst) (inst-prev i))
       (last inst i))
      ((not (in-same-basic-block? i last)))
    (if (funcall test i) (return i))))

;;;Compact SEND's into SEND2's.
(defun compact-sends (module)
  (do ((next-inst (inst-next (module-next module)) (inst-next next-inst))
       (inst (module-next module) next-inst))
      ((module? inst))
    (if (or (eq (inst-op inst) 'send) (eq (inst-op inst) 'sende))
      (if (reg? (inst-src1 inst))
        (let ((prev-inst (prev-inst-that inst #'(lambda (inst) (send-op? (inst-op inst))))))
          (if (and prev-inst (eq (inst-op prev-inst) 'send))
            (let ((new-prev-inst (prev-inst-that inst #'(lambda (i) (not (insts-commute? i inst))))))
              (when (do ((i prev-inst (inst-next i)))
                        ((eq i new-prev-inst) t)
                      (unless (insts-commute? prev-inst (inst-next i))
                        (return nil)))
                (delete-module module inst)
                (unless (eq prev-inst new-prev-inst)
                  (delete-module module prev-inst)
                  (insert-module module prev-inst new-prev-inst (inst-next new-prev-inst)))
                (setf (inst-src2 prev-inst) (inst-src1 inst))
                (setf (inst-reads prev-inst) (b+ (inst-reads prev-inst) (inst-reads inst)))
                (setf (inst-op prev-inst) (if (eq (inst-op inst) 'send) 'send2 'send2e)))))))))


;;;Return pc aligned to the next word boundary.
(defun align (pc)
  (if (oddp pc) (1+ pc) pc))

;;;Assign PC values while compacting DC's.
(defun compact-assign-pcs (module optimize-dc)
  (labels
    ((assign-pcs-sub (inst pc)
      (cond
       ((module? inst))
       ((evenp pc)
        (setf (inst-pc inst) pc)
        (assign-pcs-sub (inst-next inst) (+ pc (if (eq (inst-op inst) 'dc) 2 1))))
       ((not (and (one-elt-p (inst-predecessors inst))
                  (eq (inst-prev inst) (car (inst-predecessors inst)))))
        (assign-pcs-sub inst (1+ pc)))
       ((eq (inst-op inst) 'dc)
        (let ((prev-inst (inst-prev inst))
              (next-inst (inst-next inst)))
          (cond
           ((and optimize-dc
                 (in-same-basic-block? prev-inst inst)
                 (insts-commute? prev-inst inst))
            (swap-module module prev-inst inst)
            (assign-pcs-sub inst (inst-pc prev-inst)))
           ((and optimize-dc
                 (in-same-basic-block? inst next-inst)
                 (insts-commute? inst next-inst))
            (swap-module module inst next-inst)
            (assign-pcs-sub next-inst pc))
           (t
            (setf (inst-pc inst) (1+ pc))
            (assign-pcs-sub next-inst (+ pc 3))))))
       (t
        (setf (inst-pc inst) pc)
        (assign-pcs-sub (inst-next inst) (1+ pc))))))
    (attribute
     'pc (digraph-attributes (module-digraph module))
     (assign-pcs-sub (module-next module) 0))))


;;;Return the length of the module in words.
(defun module-length (module)
  (compact-assign-pcs module nil)
  (let ((last-inst (module-prev module)))
    (if (module? last-inst) 0
        (1+ (inst-addr last-inst)))))


;;;;+-----------------------------------+
```

```
;;;;| Expand branches into long branches. |
;;;;+-------------------------------------+

;;;Return the branch distance from the from inst to the to inst.
;;;Return nil if it cannot be determined.
(defun branch-distance (from to)
  (and (inst-pc from)
       (inst-pc to)
       (- (inst-addr to) (1+ (inst-addr from)))))

;;;Return true if a branch from the from inst would reach the to inst.
(defun branch-reaches? (from to)
  ;;Be optimistic and assume that if either pc is nil, the branch would reach.
  (let ((distance (branch-distance from to)))
    (or (null distance)
        (<= (integer-length distance) 4))))


;;;Convert short branches that do not reach their destinations into long branches.
(defun expand-branches (module)
  (labels

    ((non-reg0-conditional? (inst)
      (and (not (btest 0 (inst-live inst)))
           (or (not (equal (inst-src1 inst) reg0))
               (let ((prev (inst-prev inst))
                     (unused-reg-num (blow (bnot (inst-live inst)) 1)))
                 (if (and (< unused-reg-num 3)
                          (in-same-basic-block? prev inst)
                          (equal (inst-dst prev) reg0))
                   (let ((unused-reg (make-loc 'reg unused-reg-num)))
                     (setf (inst-dst prev) unused-reg)
                     (bclrf (inst-writes prev) 0)
                     (bsetf (inst-writes prev) unused-reg-num)
                     (setf (inst-src1 inst) unused-reg)
                     (setf (inst-reads inst) (bset unused-reg-num))
                     t))))))

     (can-change-to-shorter-branch? (inst)
      (let ((minimum-distance 15)
            minimal-dest)
        (dolist (pred (inst-predecessors (branch-dest inst)))
          (if (and (not (eq pred inst))
                   (eq (inst-op pred) 'br)
                   (or (relative? (inst-src2 pred))
                       (let ((pred2 (inst-prev pred)))
                         (and (in-same-basic-block? pred2 pred)
                              (equal (inst-src2 pred) 'reg0)
                              (eq (inst-op pred2) 'dc)
                              (setq pred pred2)))))
            (let ((distance (branch-distance inst pred)))
              (when distance
                (if (< distance 0) (setq distance (- 1 distance)))
                (when (< distance minimum-distance)
                  (setq minimum-distance distance)
                  (setq minimal-dest pred))))))
        (when minimal-dest
          (unlink-dinode inst (branch-dest inst))
          (link-dinode inst minimal-dest)
          (altered-module module)
          t)))

    (expand-sub (inst)
     (cond
      ((module? inst))
      ((or (not (branch? (inst-op inst)))
           (not (relative? (inst-src2 inst)))
           (branch-reaches? inst (branch-dest inst)))
       (expand-sub (inst-next inst)))
      ((can-change-to-shorter-branch? inst)
       (expand-sub (inst-next inst)))
      ((eq (inst-op inst) 'br)
       (simple-expand-branch inst))
      ((and (in-same-basic-block? inst (inst-next inst))
            (eq (inst-op (inst-next inst)) 'br)
            (branch-reaches? inst (branch-dest (inst-next inst))))
       (let* ((next (inst-next inst))
              (dest1 (branch-dest inst))
              (dest2 (branch-dest next)))
         (unlink-dinode inst dest1)
         (unlink-dinode next dest2)
```

```
          (reverse-condition module inst)
          (link-dinode inst dest2)
          (link-dinode next dest1)
          (altered-module module)
          (expand-sub next)))
        ((non-reg0-conditional? inst)
         (simple-expand-branch inst))
        (t
         (reverse-condition module inst)
         (insert-module module
                        (new-inst :op 'br :src2 '#.(make-loc 'rel) :live (inst-live inst))
                        inst (first (inst-successors inst)))
         (expand-sub (inst-next inst)))))

     (simple-expand-branch (inst)
      (if (btest 0 (inst-live inst))
        (error "Attempt to create a long branch with R0 live."))
      (insert-before-module module
                            (new-inst :op 'dc
                                      :src1 (make-loc 'rel)
                                      :writes '#(0)
                                      :live (bset 0 (inst-live inst)))
                            inst)
      (setf (inst-src2 inst) reg0)
      (bsetf (inst-reads inst) 0)
      (altered-module module)))

    (compact-assign-pcs module nil)
    (calc-live-regs module)
    (expand-sub (module-next module))))


;;;;+-------------------------------+
;;;;| Perform final transformations. |
;;;;+-------------------------------+
(defun inst-transformations (module)
  (insert-branches module)
  (init-vlocs module)
  (if *compact-sends* (compact-sends module))
  (attribute-steady-state
   (digraph-attributes (module-digraph module))
   (progn
     (compact-assign-pcs module *optimize-dc*)
     (expand-branches module))))


;;;;+---------------------------------+
;;;;| Miscellaneous printing functions |
;;;;+---------------------------------+

;;;Return true if the thing is a name.
(defun name? (thing)
  (or (symbolp thing) (stringp thing)))

;;Return a string containing the mapping of the character.
(defun map-char (char)
  (cond
   ((alphanumericp char) (string char))
   ((cadr (assoc char
                 '((#\_ "_")
                   (#\! "EXCLAMATION")
                   (#\$ "DOLLAR")
                   (#\% "PERCENT")
                   (#\& "AMPERSAND")
                   (#\+ "PLUS")
                   (#\- "_")
                   (#\* "TIMES")
                   (#\/ "DIVIDE")
                   (#\. "DOT")
                   (#\< "LT")
                   (#\> "GT")
                   (#\= "EQUAL")
                   (#\? "QUESTION")
                   (#\@ "AT")
                   (#\~ "NOT")
                   (#\\ "BACKSLASH")))))))

;;;Generate a string representing the name of the identifier.
;;;The name can be either a string or a symbol; if it is a string, it is passed through unaltered.
(defun make-identifier (name)
  (if (stringp name)
```

```
    name
    (apply #'concatenate
           (cons 'simple-string
                 (map 'list #'map-char (string name))))))

;;;Generate a string representing the name of the method and class.
;;; method-name is a symbol or a string containing the method name.
;;; class-name is a symbol a string containing the class name.
(defun make-module-identifier (method-name class-name)
  (if class-name
    (concatenate 'simple-string (make-identifier class-name) "__" (make-identifier method-name))
    (make-identifier method-name)))


;;;Print integer tag between 0 and 15 as a tag
(defun print-tag (tag stream)
  (format stream "~[SYM~;INT~;BOOL~;ADDR~;IP~;MSG~;CFUT~:;~:*$~X~]" tag))

;;;Print the word.
(defun print-word (word stream)
  (if (name? word)
    (write-string (make-identifier word) stream)
    (let ((a (assoc word '#.(list (list wNIL "NIL")
                                  (list wFALSE "FALSE")
                                  (list wTRUE "TRUE"))
                    :test #'equal)))
      (cond
       (a (write-string (cadr a) stream))
       ((find (tag word) '(symbol method class))
        (format stream "{~@(~A~)_~A}" (tag word) (make-identifier (data word))))
       (t
        (cond
         ((name? (tag word)) (format stream "~A:" (make-identifier (tag word))))
         ((/= (tag word) tINT)
          (print-tag (tag word) stream)
          (write-char #\: stream)))
        (cond
         ((name? (data word)) (write-string (make-identifier (data word)) stream))
         ((consp (data word))
          (format stream "~A+~D" (make-identifier (car (data word))) (cdr (data word))))
         ((eql (tag word) tINT) (format stream "~D" (data word)))
         (t (format stream "$~8,'0X" (data word)))))))))


;;;;+------------------+
;;;;| Print the module |
;;;;+------------------+

;;;Assign labels to all instructions that are destinations of branches.
(defun assign-labels (module)
  (let ((label 0))
    (all-insts
     module
     #'(lambda (inst)
         (setf (inst-label inst)
               (unless (and (one-elt-p (inst-predecessors inst))
                            (or (module? (inst-prev inst))
                                (eq (inst-prev inst) (car (inst-predecessors inst)))))
                 (incf label)))))))

;;;Print the label.
(defun print-label (num stream)
  (format stream "L~3,'0D" num))

;;;Print an addressing mode.
(defun print-loc (loc inst stream)
  (case (loctype loc)
    ((sconst lconst) (print-word (loc-num loc) stream))
    (reg (format stream "R~D" (reg-num loc)))
    (areg (format stream "A~D" (loc-num loc)))
    (sreg (prin1 (loc-num loc) stream))
    (vloc (format stream "[~D,A1]" (loc-num loc)))
    (iloc (format stream "[~D,A2]" (loc-num loc)))
    (aloc (format stream "[~D,A3]" (loc-num loc)))
    (rel (if (branch? (inst-op inst))
             (progn
               (write-char #\^ stream)
               (print-label (inst-label (branch-dest inst)) stream))
             (do ((i (inst-next inst) (inst-next i)))
                 ((and (branch? (inst-op i)) (reg? (inst-src2 i)))
                  (print-label (inst-label (branch-dest i)) stream)
```

```
                (format stream "-(*+~D)" (- (1+ (inst-addr i)) (inst-addr inst)))))))
    (t (error "Bad location ~S in ~S" loc inst))))

;;;Print the instruction.
(defun format-inst (inst stream)
  (when (inst-label inst)
    (print-label (inst-label inst) stream)
    (write-char #\: stream))
  (format stream "~:[~A~;~*~8T~]~:@(~A~)" *print-pc* #\Tab (inst-op inst))
  (let ((separator #\Tab))
    (labels
      ((print-arg (arg)
        (when arg
          (if (and *print-pc* (eql separator #\Tab))
            (format stream "~16T")
            (write-char separator stream))
          (print-loc arg inst stream)
          (setq separator #\,))))
      (print-arg (inst-src1 inst))
      (if (eq (inst-op inst) 'xlate)
        (progn
          (print-arg (inst-dst inst))
          (print-arg (inst-src2 inst)))
        (progn
          (print-arg (inst-src2 inst))
          (print-arg (inst-dst inst))))
      (if (and *print-pc* (inst-pc inst))
        (format stream "~40T;~3D~:[~;.5~]" (inst-addr inst) (oddp (inst-pc inst))))
      (terpri stream)))


;;;Print a listing of the module onto the stream.
;;; method-name is a symbol or a string containing the method name.
;;; class-name is a symbol a string containing the class name.
(defun format-module (module &optional (stream t) &key method-name class-name)
  (flet ((print-dc () (format stream "~ADC~A" #\Tab #\Tab)))
    (assign-labels module)
    (fresh-line stream)
    (format stream "MODULE ~A~%" (make-module-identifier method-name class-name))
    (print-dc)
    (print-word (make-word tMSG (cons "LoadCode" (+ 3 (module-length module)))) stream)
    (terpri stream)
    (print-dc)
    (print-word (make-word 'class class-name) stream)
    (write-char #\, stream)
    (print-word (make-word 'method method-name) stream)
    (terpri stream)
    (all-insts
     module
     #'(lambda (inst) (format-inst inst stream)))
    (format stream "~AEND~%~%" #\Tab)))
```

Front End

```
;;; -*- Mode: LISP; Base: 10.; Syntax: Common-lisp; Package: USER -*-
;;;---------------------------------------------------------------------
;;;     CST compiler
;;;     Bill Dally 8-Dec-87
;;;     Last revised 12-Jan-88
;;;     revised Andrew Chien 2/88 (various)
;;;             Waldemar Horwat 4/88 (see below)
;;;---------------------------------------------------------------------
;;;
;;;     Done:
;;;     1. Add parallel message sending
;;;     2. Add output to shell
;;;     3. Profiling
;;;     3.1. Check number of args on method invocation
;;;     3.2. Fix arg count in messages by flattening
;;;     3.3. Add distributed object creation
;;;     3.4 Add distributed object addressing
;;;     3.6 Add constituent addressing
;;;     6. Symbol and array primitive types
;;;     11. Modification of new to accept parameters
;;;     12a. Addition of a send primitive
;;;     13.  Trace functions added
;;;     14.  Default send mode is unsequenced.  SETs necessary to sequentialize
;;;     12b.  Fix send to compile to csend where appropriate
;;;     16.  Context tracing added
;;;
;;;     Adapted by Waldemar Horwat as front end for CST compiler 4/88
;;;     --Fixed two bugs that would cause emission of illegal MOVE instructions
;;;     --Removed remains of blocks and irrelevant code
;;;     --Split *constants* from *globals*
;;;     --Changed all nonessential sets into csets
;;;     --Changed all sends into csend/touch combinations
;;;       --Removed wait parameters
;;;     --Removed SEND keyword and adopted scheme-like syntax instead
;;;     --Adapted scheme-like syntax for methods
;;;

(defvar *classes* '() "Class Structure and methods")
(defvar *globals* '() "Globals and values")
(defvar *constants* '() "Constants and values")

;;;-------------------------------------
;;; Compiler front-end globals for compiling blocks
;;;
(defvar *code*)
(defvar *args*)
(defvar *vars*)
(defvar *ivars*)
(defvar *temp*)
(defvar *label*)

;;;---------------------------------------------------------------------
;;;  Front end for cst compiler
;;;---------------------------------------------------------------------

;;;---------------------------------
(defun cst-error (string &rest args)
  (apply #'format *standard-output* string args)
  nil)
(defun display-array (value)
  (let ((y nil))
    (dotimes (x (length value)) (setq y (cons (aref value x) y)))
    (format *standard-output* " ~S" (reverse y))))
;;;---------------------------------------------------------------------
;;; code relating to classes
;;;---------------------------------------------------------------------
;;  (class name ((parent-classes)) (instance-variables))
(defun compile-class (form output-stream)
  (let ((class (expand-class (cdr form))))
    (setq *classes* (cons class *classes*))
    (if output-stream (make-accessor-methods class output-stream))
    class))
;;;---------------------------------
(defun expand-class (class)
  (let ((supers (class-supers class)))
    (list (class-name class)
```

```
          supers
          (append (get-super-vars supers)
                  (cddr class))
          nil                                    ; placeholders for methods and dist
          (if (listp supers) (member 'distobj (get-superclass-list '() supers))))))
; true if distributed object
;;;---------------------------------
(defun get-superclass-list (accumulated active)
  (let ((new-acc (append active accumulated))
        (new-active (loop for class-name in active appending
                          (class-supers (get-class class-name)))))
    (setf new-active (delete '() new-active))
    (if (null new-active) new-acc
        (get-superclass-list new-acc new-active))))

;;;---------------------------------
(defun get-super-vars (supers)
  (if (and supers (listp supers) (not (eq (car supers) 'object))
           (not (eq (car supers) 'dist)))
    (append (instance-vars (car supers))
            (get-super-vars (cdr supers)))))
(defun instance-vars (class-name)
  (class-vars (get-class class-name)))
;;;---------------------------------
(defun make-accessor-methods (class output-stream)
  (let ((ivars (class-vars class))
        (class-name (class-name class)))
    (dolist (v ivars)
      (compile-method `(Method ,class-name ,v () () (return ,v)) output-stream))))
;;;---------------------------------
(defun get-class (class-name)
  (let ((class (assoc class-name *classes*)))
    (if class
      class
      (cst-error "~&Undefined Class ~S" class-name))))
(defun class-name (class) (car class))
(defun class-supers (class) (cadr class))
(defun class-vars (class) (caddr class))
;(defun class-methods (class) (cadddr class))
(defun class-dist (class) (fifth class))
;;;-----------------------------------------------------------------------
;;   (method class method-name ((args)) ((temps)) (statements))
(defun compile-method (form &optional (output-stream t))
  (if (< (length form) 6)
    (cst-error "~&Method missing field ~S" form)
    (let ((class-name (second form))
          (method-name (third form))
          (args (fourth form))
          (vars (fifth form))
          (body (nthcdr 5 form)))
      (let ((icode (compile-block args vars (instance-vars class-name) body)))
        (if output-stream
          (compile-icode method-name class-name (length args) icode :output-stream output-stream))
        icode))))

;;;-----------------------------------------------------------------------
(defun compile-block (args vars ivars body)
  ;;(format *standard-output* "~&compile-block ~S ~S ~S ~S" args vars ivars body)
  (let ((*code* nil)(*args* args)(*vars* vars)(*ivars* ivars)(*temp* 0)(*label* 0))
    (compile-statements body)
    (reverse *code*)))
;;;---------------------------------------
(defun compile-statements (body)
  (if (> (length body) 1)
    (progn (compile-statement (car body)) (compile-statements (cdr body)))
    (compile-statement `(return-x ,(car body)))))
;;
;;  Top level expressions don't require replies
;;
(defun compile-statement (stat)
  (compile-expression '() stat))
;;;---------------------------------------
(defun symbol-is-keyword? (expr)
  (find expr
        (if *anachronisms*
          '(set cset reply reply-x forward exit iftrue if begin
            new newco quote send msg touch)
          '(set cset return return-x reply reply-x exit iftrue if begin
            new newco quote msg touch))))

(defun symbol-type (expr)
```

```
  (cond ((numberp expr) `(const ,expr))
        ((not (symbolp expr)) nil)
        ((eq expr 'self) 'self)
        ((eq expr 'super) 'super)
        ((eq expr 'group) 'group)
        ((member expr *vars*) `(var ,(index expr *vars*)))
        ((member expr *args*) `(arg ,(index expr *args*)))
        ((member expr *ivars*) `(ivar ,(index expr *ivars*)))
        ((symbol-is-keyword? expr) 'keyword)
        ((assoc expr *globals*) `(global ,expr))
        ((assoc expr *constants*) `(const ,(cdr (assoc expr *constants*))))
        (t (list 'method expr))))

(defun compile-atom (slot expr)
  (let ((type (symbol-type expr)))
    (if type
      (check-binding slot type)
      (cst-error "~&bad atomic expression ~S" expr))))

;;;---------------------------------------
;;; compiles an expression and puts the result in slot
;;; if slot is nil, doesn't put the result anywhere.
;;; if slot is '_unbound_ creates a temporary
(defun compile-expression (slot expr)
  ;(format *standard-output* "~&compile-expression ~S ~S" slot expr)
  (if (atom expr)
    (compile-atom slot expr)
    (let ((head (car expr)))
      (if (eq (symbol-type head) 'keyword)
        (ecase head
               ((set cset) (compile-set slot expr))
               ((return return-x) (compile-return head slot expr))
               (reply
                (if *anachronisms* (compile-return 'return slot expr)
                    (compile-reply 'reply slot expr)))
               (reply-x
                (if *anachronisms* (compile-return 'return-x slot expr)
                    (compile-reply 'reply-x slot expr)))
               (forward ;anachronism
                (if (eq (cadr expr) 'requester)
                  (compile-reply 'reply-x slot (list 'reply-x (cddr expr)))
                  (cst-error "~&Can't reply to ~S" (cadr expr))))
               (exit (emit '(exit)) slot)
               (iftrue (compile-iftrue slot expr))
               (if (compile-if slot expr))
               (begin (compile-begin slot expr))
               (new (compile-new slot expr))
               (newco (compile-newco slot expr))
               (quote (check-binding slot `(const ,(cadr expr))))
               (msg (compile-message slot expr))
               (send ;anachronism
                (compile-expression slot (cdr expr)))
               (touch (compile-touch slot expr)))
        (compile-send slot expr)))))
;;;---------------------------------------
(defun compile-begin (slot expr) (compile-begin-1 slot (cdr expr)))
(defun compile-begin-1 (slot expr)
  (if (> (length expr) 1)
    (progn (compile-statement (car expr)) (compile-begin-1 slot (cdr expr)))
    (compile-expression slot (car expr))))
;;;---------------------------------------
;;  (new class-name)
;;  (new class-name init-parameter)
(defun compile-new (slot expr)
  (let* ((tl (check-bound slot))
         (class-name (cadr expr)))
    (if (= 2 (length expr))
      (emit `(new ,tl ,class-name))
      (emit `(new ,tl ,class-name ,(compile-atom '_unbound_ (third expr)))))
    tl))
;;;---------------------------------------
(defun lookup-writeable-id (name)
  (let ((obj (compile-atom '_unbound_ name)))
    (if (and (listp obj) (member (car obj) '(var ivar global)))
      obj
      (cst-error "~&can't set identifier ~S" name))))
;;;---------------------------------------
;;; special for to make a constituent of a distributed object
;;     (newco node-number newindex DO-size root)  ;; only used by distobj def
(defun compile-newco (slot expr)
  (if (check-length 5 expr)
```

```
    (let* ((tl (check-bound slot))
           (args (mapcar #'(lambda (x) (compile-expression '_unbound_ x)) (cdr expr))))
      (emit `(newco ,tl ,@args))
      tl)))
;;;---------------------------------------
;;  (set slot value-expression)
;;  (cset slot value-expression)
(defun compile-set (slot expr)
  (if (check-length 3 expr)
    (let* ((dest (lookup-writeable-id (cadr expr)))
           (type (car expr))
           (wait (eq type 'set)))
      (compile-expression dest (caddr expr))
      (if wait (emit `(touch ,dest)))
      (check-binding slot dest))))
;;;---------------------------------------
;;   (reply value-expression)
;;   (reply-x value-expression)
(defun compile-reply (head slot expr)
  (if (check-length 2 expr)
    (let ((result (compile-expression '_unbound_ (cadr expr))))
      (emit `(,head ,result))
      (check-binding slot result))))
;;;---------------------------------------
;;   (return value-expression)
;;   (return-x value-expression)
(defun compile-return (head slot expr)
  (if (check-length 2 expr)
    (let ((result (compile-expression '_unbound_ (cadr expr))))
      (emit `(,head ,result))
      (check-binding slot result))))
;;;---------------------------------------
;;   (selector dest (args))
(defun compile-send (slot expr)
  (let ((selector (compile-expression '_unbound_ (first expr)))
        (dest (check-bound slot))
        (args (mapcar #'(lambda (x) (compile-expression '_unbound_ x)) (cdr expr))))
    (if (eq (car selector) 'const)
      (cst-error "~&can't send to ~S" (first expr)))
    (emit `(csend ,dest ,selector ,@args))
    dest))
;;;---------------------------------------
;;  (msg node# selector dest (args)) not implemented
(defun compile-message (slot expr)
  (let ((args (mapcar #'(lambda (x) (compile-expression '_unbound_ x)) (cdr expr))))
    (emit `(msg ,@args))
    slot))
;;;---------------------------------------
;;  (iftrue cond-expression true-block false-block)
(defun compile-iftrue (slot expr)
  (if (< (length expr) 3)
    (cst-error "~&syntax error ~S" expr)
    (let* ((l1 (new-label))
           (t1 (compile-expression '_unbound_ (cadr expr)))
           (t2 (check-bound slot))
           (c1 (compile-expression '_unbound_ (caddr expr)))
           (c2 (if (> (length expr) 3)
                 (compile-expression '_unbound_ (cadddr expr))
                 nil)))
      (emit `(falsejump ,t1 ,l1))
      (emit `(csend ,t2 value ,c1))
      (if c2
        (let ((l2 (new-label)))
          (emit `(jump ,l2))
          (emit `(label ,l1))
          (emit `(csend ,t2 value ,c2))
          (emit `(label ,l2)))
        (emit `(label ,l1)))
      t2)))
;;;---------------------------------------
;;  (if condition-expression true-arm false-arm)
(defun compile-if (slot expr)
  (if (< (length expr) 3)
    (cst-error "~&syntax error ~S" expr)
    (let* ((l1 (new-label))
           (t1 (compile-expression '_unbound_ (cadr expr)))
           (t2 (check-bound slot)))
      (if (and (= (length expr) 3) t2) (emit `(move ,t2 (const nil))))
      (emit `(falsejump ,t1 ,l1))
      (compile-expression t2 (caddr expr))
      (if (> (length expr) 3)
```

```
        (let ((l2 (new-label)))
          (emit `(jump ,l2))
          (emit `(label ,l1))
          (compile-expression t2 (cadddr expr))
          (emit `(label ,l2)))
        (emit `(label ,l1)))
      t2)))
;;;---------------------------------------
;;  (touch variable)
(defun compile-touch (slot expr)
  (when (check-length 2 expr)
    (if (null slot) (setq slot '_unbound_))
    (let ((dest (check-binding slot (compile-expression slot (second expr)))))
      (emit `(touch ,dest))
      dest)))
;;;---------------------------------------
(defun check-length (n expr)
  (if (/= (length expr) n)
    (cst-error "~&syntax error ~S" expr)
    t))
;;;---------------------------------------
(defun check-bound (a) (if (eq a '_unbound_) (new-temp) a))
;;;---------------------------------------
;;; if a is already bound move b to a and return a otherwise return b
(defun check-binding (a b)
  (if (eq a '_unbound_)
    b
    (if (equal a b)
      a
      (progn (if a (emit `(move ,a ,b)))
             a))))
;;;---------------------------------------
(defun new-label ()
  (let ((result *label*))
    (setq *label* (+ *label* 1))
    result))
;;;---------------------------------------
(defun new-temp ()
  (let ((result `(temp ,*temp*)))
    (setq *temp* (+ *temp* 1))
    result))
;;;---------------------------------------
(defun emit (code)
  (setq *code* (cons code *code*))
  ;(format *standard-output* "~&emit ~S" code)
  )
;;;---------------------------------------
(defun index (a l) (index1 a l 0))
(defun index1 (a l n)
  (if l (if (eq a (car l)) n (index1 a (cdr l) (+ 1 n)))))
;;;-------------------------------------------------------------------------
(defun compile-global (form)
  (push (cons (cadr form) (caddr form)) *globals*))
;;;-------------------------------------------------------------------------
(defun compile-constant (form)
  (push (cons (cadr form) (caddr form)) *constants*))
```

Compiler

```
;;;
```

```
(defparameter *use-lru-register-allocation* t)

;Keep track of variables in the registers during code generation and use values from the
;registers instead of from memory whenever possible.
(defparameter *optimize-local-regs* t)

;Keep track of whether it is possible for the instance object to have migrated away.
;Don't force it if it could not have migrated away.
(defparameter *optimize-migrate* t)

;Don't XLATE the instance object if there are no references to it.
(defparameter *optimize-ivar-access* t)

;Don't allocate a context unless it is actually used.
(defparameter *optimize-null-contexts* t)

;Send message to the current node if the receiver is SELF and it is not atomic.
(defparameter *optimize-send-self* t)

;Try to combine SENDs and SENDEs into SEND2s and SEND2Es.
(defparameter *compact-sends* t)

;Try to align DC's on word boundaries whenever possible.
(defparameter *optimize-dc* t)


;Use an additional reply node field in SENDs.
(defconstant *reply-node* nil)

;Print program counter values as comments in output.
(defparameter *print-pc* t)

;;;Set all optimizations to value.
(defun all-optimizations (value)
  (setq *delete-dead-defs* value)
  (setq *delete-moves* value)
  (setq *delete-touches* value)
  (setq *dflow-optimizations* value)
  (setq *fold-constants* value)
  (setq *forward-sends* value)
  (setq *merge-code* value)
  (setq *optimize-primitives* value)
  (setq *optimize-forces* value)
  (setq *optimize-vars* value)
  (setq *reg-variables* value)
  (setq *use-lru-register-allocation* value)
  (setq *optimize-local-regs* value)
  (setq *optimize-migrate* value)
  (setq *optimize-ivar-access* value)
  (setq *optimize-null-contexts* value)
  (setq *optimize-send-self* value)
  (setq *compact-sends* value)
  (setq *optimize-dc* value))



;;;;+----------------------+
;;;;| Debugging parameters |
;;;;+----------------------+
;;;Warning:  Do not change this setting without recompiling all files!

;Use debugging data structures, which results in easier debugging but a slower compiler.
(defvar *debug* t)


#+lispm (import 'si:loop-tassoc)

#+:Macintosh (set-mac-default-directory "HDrive:MDP:Compiler:")
(load "Utilities")
(load "Word")
(load "Stmt")
(load "Inst")
(load "GenStmt")
(load "GenInst")
(load "GenAsm")

;;;Compile the icode for the method and output it on the stream.
;;; nargs is the number of arguments for the method.
;;; method is a symbol or a string containing the method name.
;;; class is the class.
;;; stream is the stream onto which the output is to be printed.
```

```
(defun compile-icode (method class nargs icode &key (output-stream t))
  (when *verbose-cst*
    (format t "~&;Compiling and optimizing I-Code for method ~S ~S...~%" class method))
  (let ((stmtgraph (optimize-stmtgraph (input-icode icode))))
    (when *verbose-cst*
      (format t "~&;Compiling to assembly language method ~S ~S...~%" class method))
    (let ((module (compile-stmtgraph stmtgraph nargs (class-nivars class))))
      (inst-transformations module)
      (format-module module output-stream
                     :method-name method
                     :class-name class))))

;;;Compile the icode and output the resulting stmtgraph.
;;;This function is for debugging purposes only.
(defun optimize-icode (icode)
  (optimize-stmtgraph (input-icode icode)))


(load "Front")


(defconstant atomic-classes '(Integer Symbol Boolean Float))
;;;Return the number of instance variables the class has or nil if the class is atomic.
(defun class-nivars (class)
  (unless (find class atomic-classes)
    (length (instance-vars class))))


;;;;+----------+
;;;;| Compiler |
;;;;+----------+

;;;Compile the method and output the optimized icode and the number of variables used.
;;;This function is for debugging purposes only.
(defun optimize-method (form)
  (optimize-icode (compile-method form nil)))


;;;Compile the form to the output-stream.
(defun compile-form (form &optional (output-stream t))
  (when *verbose-cst*
    (format t "~&;Compiling ~S ~S~@[ ~S~]...~%" (car form) (cadr form)
            (if (eq (car form) 'method) (caddr form))))
  (let ((head (car form)))
    (case head
      (class (compile-class form output-stream))
      (method (compile-method form output-stream))
      (global (compile-global form))
      (constant (compile-constant form))
      (load (compile-from-file (cadr form) output-stream))
      (t (error "Bad form: ~S" form)))))

;;;Compile the forms to the output-stream.
(defun compile-forms (forms &optional (output-stream t))
  (dolist (form forms)
    (compile-form form output-stream)))

;;;Compile from the file named in-file-name to the output-stream.
(defun compile-from-file (in-file-name &optional (output-stream t))
  (with-open-file (in-file in-file-name :direction :input)
    (do ((x (read in-file nil 'exit) (read in-file nil 'exit)))
        ((eq x 'exit))
      (compile-form x output-stream))))

;;;Initialize the cst classes, globals, and constants.
(defun init-cst ()
  (setq *classes* nil)
  (setq *globals* nil)
  (setq *constants* nil)
  (compile-forms '((Class Object ())
                   (Class Integer (Object))
                   (Class Float (Object))
                   (Class Symbol (Object))
                   (Class Boolean (Object))
                   (Class Array (Object) 1 a)
                   (Constant nil nil)
                   (Constant t true)
                   (Constant true true)
                   (Constant false false)) nil))

(init-cst)
```

```
;;;Compile from the file named in-file-name to the file named out-file-name.
(defun cst (in-file-name out-file-name)
  (init-cst)
  (with-open-file (out-file out-file-name :direction :io :if-exists :supersede)
    (compile-from-file in-file-name out-file)))
```

Appendix E. Using the Compiler

In order to use the compiler, first load the LOOP macro and install it with (use-package 'loop) unless it was already installed. Then load the Compiler.lisp or a compiled Compiler file, which should automatically load all other compiler files. At this point the compiler can be used interactively or to compile entire files. The available calls are as follows:

Compiling Files:

- (cst input-file-name output-file-name) will compile the file named input-file-name and write the compiled methods to a new file named output-file-name. Progress information and errors, if any, are printed to the terminal.

Interactive Compilation:

- (init-cst) erases the compiler's knowledge of previous user-defined classes. It is automatically called by the cst function.

- (compile-forms forms-list output-file-stream) compiles the Concurrent Smalltalk forms given as a list in forms-list and writes the resulting code onto output-file-stream. If output-file-stream is omitted, the code is written to the terminal. This method does not call init-cst, thereby allowing interactive compilation.

- (compile-form form output-file-stream) compiles the single Concurrent Smalltalk form given as a list in form and writes the resulting code onto output-file-stream. If output-file-stream is omitted, the code is written to the terminal; if it is `nil`, no code is generated. If the form is a method definition, this method returns the I-Code generated by the Front End when compiling the form. This method does not call init-cst, thereby allowing interactive compilation.

- (all-optimizations state) turns all Optimist optimizations that can be disabled off (if state is `nil`) or on otherwise.

Bibliography

[1] Abelson, Harold and Sussman, Gerald J. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, 1985.

[2] Aho, Alfred V., Sethi, Ravi, and Ullman, Jeffrey D. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[3] Burke, Glenn and Moon, David. "Loop Iteration Macro." MIT Laboratory for Computer Science Memo TM-169, January 1981.

[4] Chaitin, G. J. "Register Allocation and Spilling via Graph Coloring." *ACM SIGPLAN Notices* **17:6**, 1982.

[5] Dally, William J. *A VLSI Architecture for Concurrent Data Structures*. Kluwer, Boston, MA, 1987.

[6] Dally, William J. et al. "Architecture of a Message-Driven Processor." *Proceedings of the 14th ACM/IEEE Symposium on Computer Architecture*, June 1987, pp. 189-196.

[7] Dally, William J. and Chien, Andrew A. "Object-Oriented Concurrent Programming in CST." *Proceedings of the 3rd Conference on Hypercube Concurrent Computers and Applications*, Pasadena, CA, January 1988.

[8] Goldberg, Adele and Robson, David. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, MA, 1983.

[9] Horwat, Waldemar and Totty, Brian. "Message-Driven Processor Architecture, Version 10." MIT Concurrent VLSI Architecture Memo, March 1988.

[10] Horwat, Waldemar and Totty, Brian. "Message-Driven Processor Simulator, Version 5.0." MIT Concurrent VLSI Architecture Memo, December 1987.

[11] Steele, Guy L. *Common Lisp: The Language*. Digital Press, Digital Equipment Corporation, 1984.

[12] Totty, Brian. "An OS Kernel for the Jellybean Machine, Version 1.0." MIT Concurrent VLSI Architecture Memo, August 1987.

[13] Wulf, William M., Johnsson, Richard K., Weinstock, Charles B., Hobbs, Steven O., and Geschke, Charles, M. *The Design of an Optimizing Compiler*. American Elsevier, New York, 1975.